Deliverable-2.6: RINA simulator

advanced functionality incorporating use-case specific models

Deliverable Editor: Vladimir Vesely, Faculty of Information Technology, Brno University of Technology (FIT-BUT)

Publication date:	7-December-2015
Deliverable Nature:	Software/Report
Dissemination level (Confidentiality):	PU (Public)
Project acronym:	PRISTINE
Project full title:	PRogrammability In RINA for European Supremacy of virTualIsed NEtworks
Website:	www.ict-pristine.eu
Keywords:	RINA Simulator, OMNeT++, simulation models
Synopsis:	This document describes RINASim, your discrete event simulation framework of native RINA networks.

The research leading to these results has received funding from the European Community's Seventh Framework Programme for research, technological development and demonstration under Grant Agreement No. 619305.

Copyright © 2014-2016 PRISTINE consortium, (Waterford Institute of Technology, Fundacio Privada i2CAT - Internet i Innovacio Digital a Catalunya, Telefonica Investigacion y Desarrollo SA, L.M. Ericsson Ltd., Nextworks s.r.l., Thales Research and Technology UK Limited, Nexedi S.A., Berlin Institute for Software Defined Networking GmbH, ATOS Spain S.A., Juniper Networks Ireland Limited, Universitetet i Oslo, Vysoke ucenu technicke v Brne, Institut Mines-Telecom, Center for Research and Telecommunication Experimentation for Networked Communities, iMinds VZW.)

List of Contributors

Deliverable Editor: Vladimir Vesely, Faculty of Information Technology, Brno University of Technology (FIT-BUT)
fit-but: Vladimir Vesely, Tomas Hykel, Marcel Marek, Ondrej Rysavy, Jerabek Kamil, Ondrej Lichtner
i2cat: Eduard Grasa
upc: Sergio Leon Gaixas
uio: Peyman Teymoori
tssg: Micheal Crotty

Disclaimer

This document contains material, which is the copyright of certain PRISTINE consortium parties, and may not be reproduced or copied without permission.

The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the PRISTINE consortium as a whole, nor a certain party of the PRISTINE consortium warrant that the information contained in this document is capable of use, or that use of the information is free from risk, and accept no liability for loss or damage suffered by any person using this information.

Executive Summary

Simulation is widely accepted validation and verification tool to test and prove new technologies. Simulation runs can reveal design flaws, performance drawbacks and other weak points. Simulation results are able to enhance development process of researchers and programmers. Hence, the implementation of the Recursive Internet Architecture Simulator (RINASim) is a natural step to support the design and development of the RINA SDK.

RINASim is independent full-fledged simulation framework of native RINA networks for OMNeT++ discrete event simulator. RINASim allows its user to inspect RINA behavior in different deployment topologies. Moreover, RINASim offers flexible development of new policies, which may directly impact and alter interprocess communication. Furthermore, RINASim has easily extensible statistic collection system, which provides accurate results gathering and evaluation.

Previous Deliverable 2.4 outlined RINASim basic functionality. Since that time, RINASim has significantly matured in terms of both width and depth of its functionality. This document provides detailed RINASim user guide together with a description of basic RINA principles that influenced RINASim design and development.

Table of Contents

List of Figures

1. Introduction

This deliverable provides the specification, design and implementation details of RINASim - Simulator of RINA implemented in OMNeT++ tool. The aim of this report is to provide comprehensive information on RINASim helping researchers and practitioners to understand the underlying concepts and to utilize the simulator in their experiments.

RINA presents a new approach to network architecture and as such the extra information and supporting tools should be provided to understand fully various concepts included. Chapter 2 denotes a fundamental theory behind RINA, which served as the cornerstone for RINASim development. The presentation starts with a discussion on the character of applications in RINA environment. RINA applications run as processes that utilize network through application entities (AE). Each AE employs communication protocol to govern data transfer and controls necessary internetworking tasks, which stands for the interprocess communication (IPC). Processes that can establish IPC are organized in a Distributed IPC Facility (DIF), which represents a layer in RINA. Next, the following core principles of RINA are discussed:

- differences between the connection-oriented or connectionless style of communication and their impact on data transport in RINA,
- the role of Delta-T protocol on the design of communication patterns in RINA, mainly the importance of Delta-T for design protocols with soft-state,
- identification of mechanisms and policies, where former stands for fixed functionality of IPCS, while latter specifies additional features, and
- the naming and addressing model introduced with RINA.

RINASim is a simulator developed in OMNeT tools, which is one of the most used networking simulators today. To efficiently use RINASim one needs to know the basics of OMNeT at least. While RINASim consists of predefined simulation modules for many of RINA concepts and policies, to fully exploit the simulation environment C programming skills are necessary too. Chapter 3 contains information on the installation of OMNeT, acquiring RINASim from public GitHub repository and installing it in OMNeT++. Also, the reader will get information on running the simulation of RINA models provided together with the RINASim. The information presented in Chapter 3 should be sufficient for RINASim novice to start with the simulator as a tool for learning RINA or doing research in networking, respectively.

Chapter 4 provides a high-level concept overview of RINA DIF and DAF parts and their interaction. This overview serves to identify the key concepts that are delivered in the form of simulation components and models in RINASim. The presentation follows a top-down approach, discussing RINA nodes first, then moving focus to DAF and DIF design. In DIF

design, all important mechanisms are described. They consist of enrollment that takes places when IPCP joins the existing DIF. Then, data transfer that covers data delimiting, error and flow control, their relaying and multiplexing is presented. The functions of flow allocator and resource allocator are specified. These two components are important for setting necessary resources to establish a flow between two endpoints. Information on respective end points of the flow is taken from RIB served by RIB daemon. Finally, an overview of the important properties of Common Distributed Application Protocol (CDAP) is given. The presented information serves as a foundation for the design of RINASim components presented in next.

Chapter 5 thoroughly describes all the implementation specifics of available RINASim simulation modules. The design of RINASim architecture was driven by the requirement for the tight correspondence between the structure of the RINA specification and proposed simulation model. While the tight correspondence may not lead to an efficient implementation of RINA stack, for simulation model this does not represent an issue. Contrary, the correspondence between the specification structure and the simulation model makes the understading easier for the users. The specification was transformed to simulation model following the well-defined design template, which provides necessary information to anyone who wish to extend the RINASim with new mechanisms or policies. Thus, Chapter 5 is the ultimate source of information for RINASim contributors.

RINA introduces policies as a way to specified additional features or optional functionality. Chapter 6 briefly describes currently implemented policies in RINASim that are related to Flow Allocation, EFCP, Resource Allocation, Relay and Multiplexing functionality and Routing. Provided information documents each policy by describing its purpose, specifying and explaining parameters and localizing the policy implementation in the source code.

One of the purposes of RINASim is to support research on various RINA policies. The role of the simulator is mainly in the evaluation phase. Chapter 7 provides information about policy-driven advanced behavior. In this chapter, models implementing congestion avoidance and control, scheduling, routing and forwarding policies are documented.

The other intention of RINASim is to provide a tool for carrying out experiments using simulator scenarios. Chapter 8 contains Demo scenario together with three experimental setups. Each setup is defined in terms of network topology definition, description of network functions and behavior accompanied by simulation configuration files. Possible users of RINASim may find the information presented in this chapter interesting. The demo scenario is a comprehensive guideline on the usage of RINASim simulator.

RINASim evolved from a simple simulator to a complex simulation environment that implements many of RINA mechanisms and policies. It is available from the GitHub repository and run in the current OMNeT++ environment. It can be used for research of RINA Policies

as well as for evaluation of various network scenarios. This deliverable contains information on design and implementation of RINASim. Next it provides the guideline for installation and deploying RINASim. Finally, the present deliverable demonstrates the utilization of RINASim on three network scenarios.

2. Brief Theory

The purpose of this chapter is to provide future RINASim user with a short introduction to RINA concepts. These concepts and ideas formulated the design and development of the whole RINASim. Others may consider this chapter as a useful source of condensed information about RINA.

2.1. Nature of applications and application protocols

Is application a part of IPC environment or not? 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 and also one application may have many application protocols.

Following terms are recognized in the frame of RINA, and their relationship is depicted in 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.

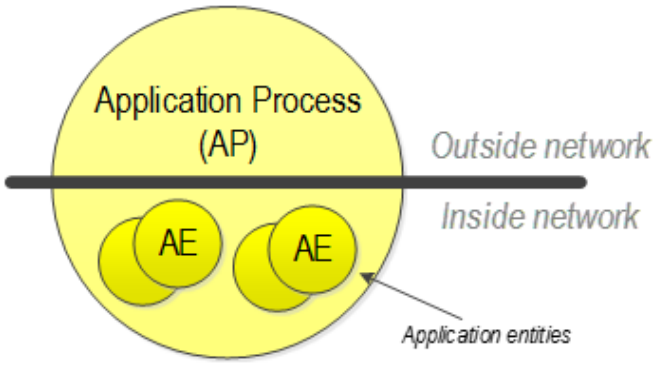

Figure 1. Application Protocol and Application Entities relationship

There may be multiple instances of the Application Process in the same system. AP may have multiple AEs, each one may process 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 SDUs.

2.2. 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. As RINA name suggests, 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). 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. RINA is based on following core ideas:

> — "Networking is interprocess communication…and IPC only!" [rina-intro]

> — "Application Processes communicate via a service provided by a distributed application that provides IPC. The application processes that make up this Distributed IPC Facility provide a protocol that implements an IPC mechanism, and a protocol for managing distributed IPC (routing, security and other management tasks)." [networking-is-ipc]

In ISO/OSI or TCP/IP, there is a set of layers each with completely different functions. RINA on the other hand yields 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 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. **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 in figure below:

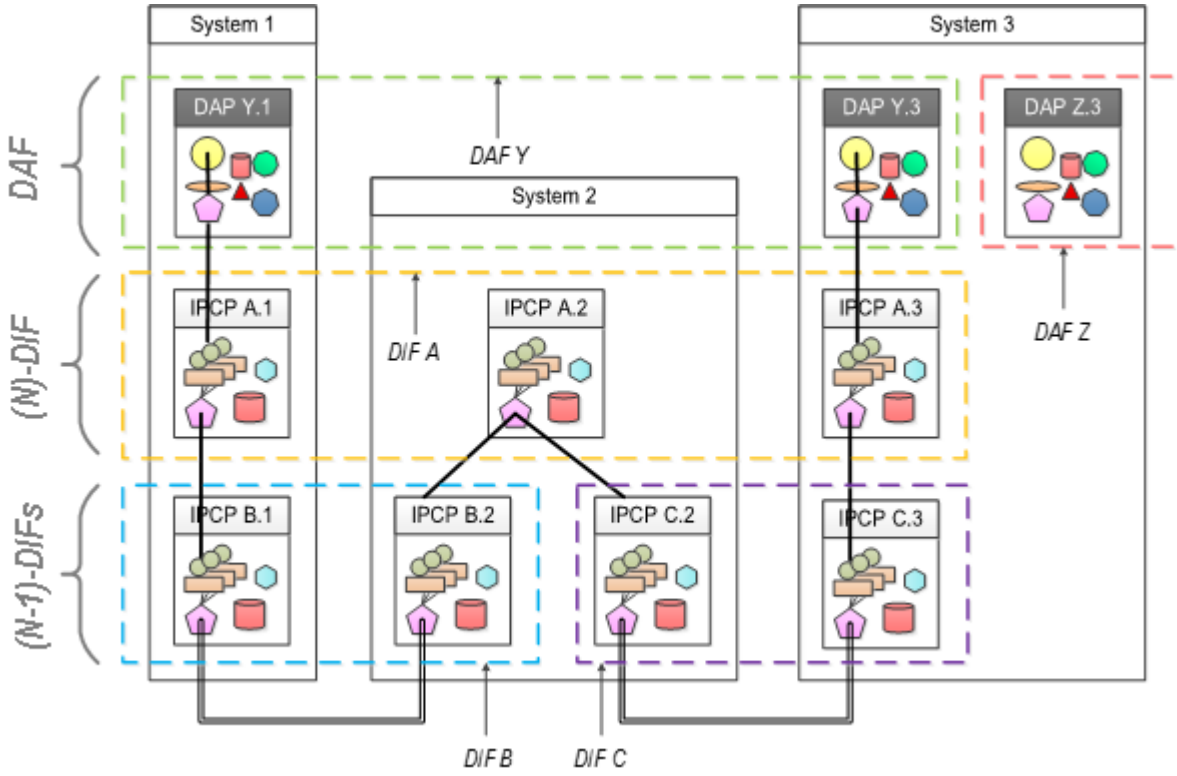

Figure 2. DIF, DAF, DAP and IPCP illustration

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., data-link 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.

2.3. Connection-oriented vs. connection-less

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 layer determines what mechanisms and policies to use. Allocation is accompanied with access rights and

description of QoS demands (e.g., what minimum bandwidth or delay is needed for correct operation of application).

2.4. 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 Delta-t protocol (see [delta-t-spec] and [delta-t-features]). 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 PDU for retransmission while waiting for a positive acknowledgment (a.k.a. R-timer); c) **Maximum time before Acknowledgement** (a.k.a. A-timer).

Delta-t assumes that all connections exist all the time. Synchronization state is maintained only during the activity, but after 2-3 MPL periods without any traffic it 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.

2.5. Separation of mechanism and policy

We understand term mechanism as the fixed part and policy as the flexible part of IPC. Just to remind the reader that mechanism is fixed, the policy is flexible part of any 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 handle 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 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 on each other.

This implies that only 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 state internal to its PM, where application protocol (carried inside) modifies state external to PM.

2.6. Naming and addressing

Application Process communicates in order to share state. 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 **Application Process Name (APN)** as globally unambiguous, location-independent, system-dependent name. **Application Process Instance Identifier (API-id)** differentiates between multiple instances of the same AP in the system. **Application Entity Instance Identifier (AEI-id)**, which is unambiguous for a single AP, helps us to identify different AE instances of same **Application Entity Name (AEN)** within AP. **Application Naming Information (ANI)** references a complete set of identifiers to name particular application; it consists of four-tuple APN, API-id, AEN, and AEI-id. The only required part of ANI is APN; others are optional. Distributed Application Name (DAN) is globally unambiguous name for a set of system-independent APs.

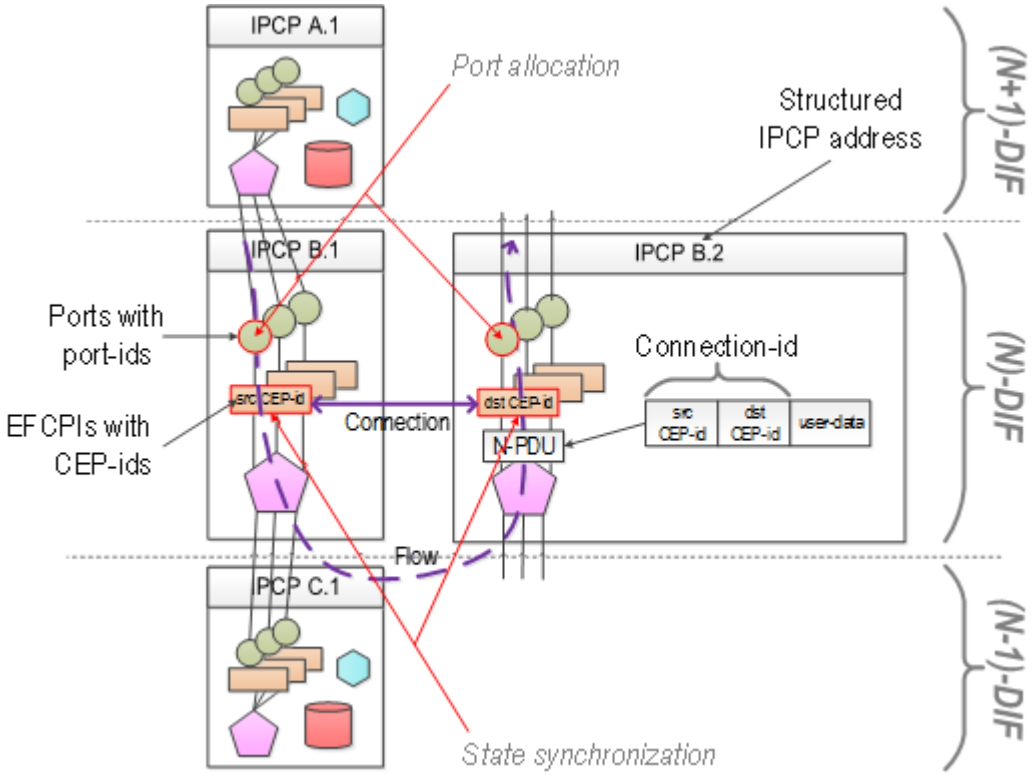

Figure 3. IPCP local identifiers overview

IPC Process has APN to identify it among other DIF members. An 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 below 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 N-PMs – 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 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. Installation and configuration

The section explains how to install, configure and deploy the RINASim environment.

RINASim installation is a straightforward process with two phases: 1) obtain the project; 2) compile the project, which creates one static library (`librinasimcore` containing simulation core) and one dynamic library (`librinasim` also containing various policies linked together with core). Nevertheless, this tutorial will dive into details regarding installation and setup process.

3.1. Support

FIT-BUT provides support for the current developer master branch version. Users can:

1. contact developers via mail (comment headers of source code should contain the author's email);
2. try to post problems as a new tickets via [ops-rinasimtickets] webpage;
3. join shared developers Skype group chat and send him/her message (just past the following text into Skype `skype:?chat&blob=ucdWTg4wJEILgDahhm9tTuUxGQ8Yr3F2UJTH-n6lE8qVZfOJKdVUREJ4YyTb91lKEZ3JoOgS9biF003e`);
4. use official RINASim mailing list and join rinasim@fit.vutbr.cz[1];

3.2. OMNeT Installation

RINASim is developed in OMNeT 4.6, but its source codes are fully backward compatible with older (i.e., 4.5) and also newer (i.e., 5.0) 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 [github-kvetak]. Apart from this official channel, RINASim stable release snapshots are periodically published on Open Source Project repository [ops-rinasim].

3.2.1. Windows Installation

1. Download source codes from the official web pages [omnetpp-dwnld]. Beware that in a case of 64-bit platform, the simulator, and its libraries are still compiled for a 32-bits architecture.

[1] mailto:rinasim@fit.vutbr.cz

2. Unpack the source code archive. Preferably to a folder residing on the hard disk root (like *C:\omnetpp-45*).
3. Execute the `mingwenv.cmd` program.
4. In an open MinGW prompt, type `./configure`. Check whether you have all the prerequisites.
5. Execute `make`, then wait until the whole project successfully builds itself.
6. Run OMNeT++ IDE from MinGW prompt by typing `omnetpp`, or use shortcut in *<install-dir>\ide\omnetpp.exe*
7. If you plan to run outside IDE simulations, then you have to add *<install-dir>\bin* to the `PATH`.

3.2.2. Linux installation

1. Among prerequisities are the following packages: `build-essential gcc g++ bison flex perl tcl-dev tk-dev libxml2-dev zlib1g-dev default-jre doxygen graphviz libwebkitgtk-1.0-0 openmpi-bin libopenmpi-dev libpcap-dev`
2. Download source codes from the official webpages [omnetpp-dwnld].
3. Unpack the source code archive with `tar xvfz omnetpp-4.6-src.tgz`.
4. Type `. setenv` to add the directory to PATH.
5. Execute `./configure && make`, then wait until the whole project successfully builds itself.
6. Optionally create shortcuts by running `make install-menu-item` and `make install-desktop-icon`
7. Run the OMNeT IDE by typing `omnetpp` or using shortcut.

3.3. RINASim Installation

The reader is advised to clone one of the following repositories containing RINASim:

- Latest official stable release on OpenSourceProjects repository:

```
git clone https://opensourceprojects.eu/git/p/pristine/rinasimulator/rinasim rinasim
```

- Current developers master branch, which should always contain runnable code:

```
git clone https://github.com/kvetak/RINA.git rinasim
```

Once you have any version of RINASim source codes then you can start with RINASim installation:

3.3.1. The IDE way

1) Open the OMNeT IDE and start project import, menu item *File* → *Import…*.

2) Choose *General* and option *Existing Projects into workspace*.

3) Depending on the form of your source codes, choose either *Select root directory* or *Select archive file*.

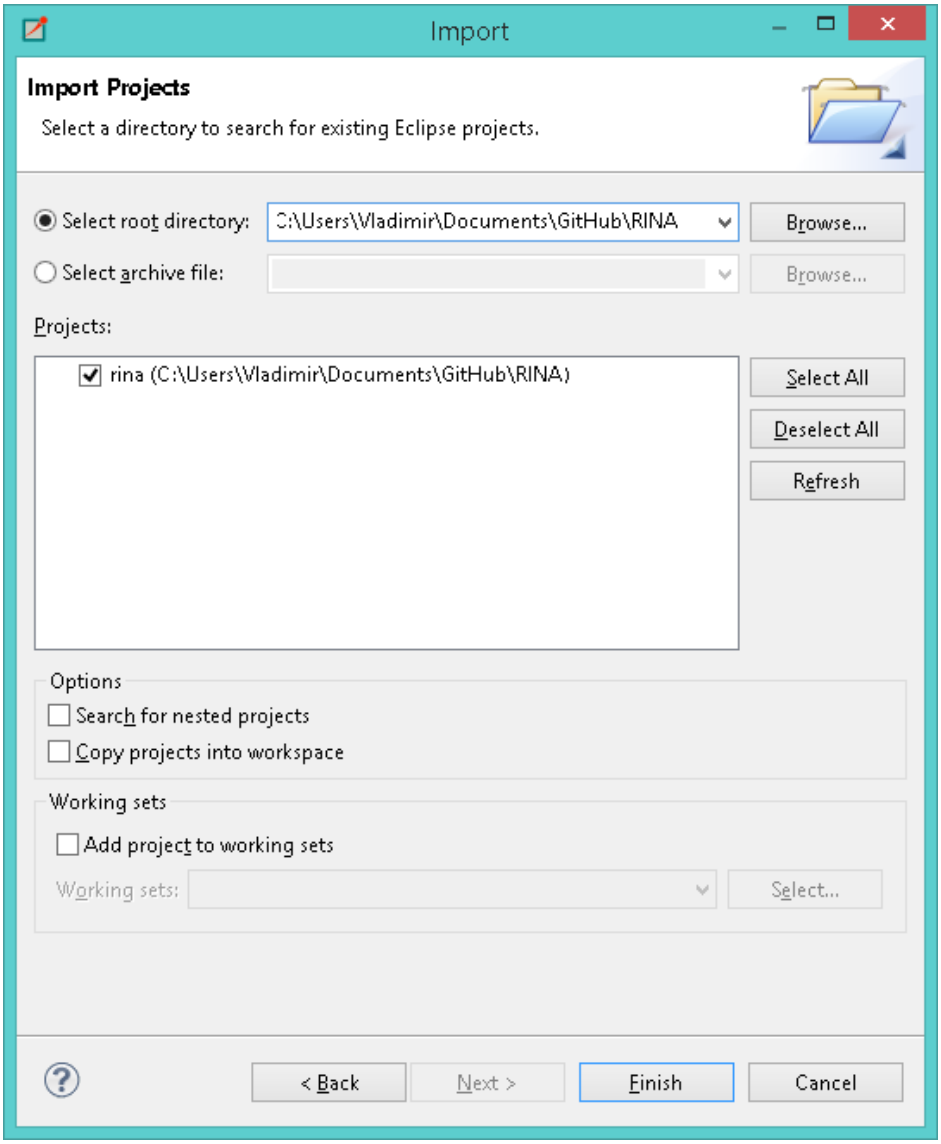

Figure 4. Import Wizard

4) Conclude import via the *Finish* button. Now RINASim should be available in the Project Explorer under folder `rina`

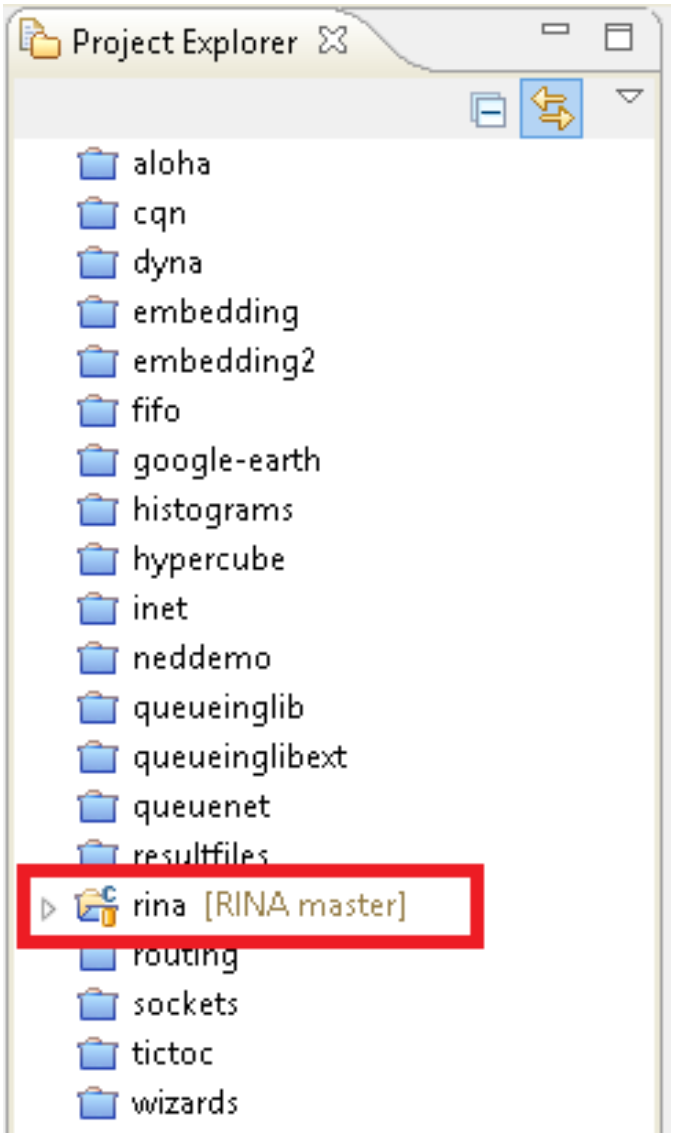

Figure 5. Project Explorer

5) Build the `rina` project by *Project* → *Build project*.

3.3.2. The command line way

1) Prepare the console environment:

- on Windows: Execute the `mingwenv.cmd` batch file inside the OMNeT++ folder.
- On UN*X platforms: Open a console, navigate to the OMNeT++ folder and run `. ./setenv`.

2) Enter the root directory of RINASim.

3) Build RINASim by invoking `make`.

3.3.3. Makefile

RINASim source code is split between policies (folder with the same name) and simulator core (folder `src`). We have removed a circular dependency between these folders. This allowed RINASim source codes to compile based on two automatically created Makefiles (in `policies` and `src`) and one master `Makefile` (in the root). Thanks to that, developers should not experience random rebuilds of the whole project now. Currently, compiling `src` folder creates static library `librinasimcore.a`, which contains only RINASim core without policies. Subsequently, `policies` folder is compiled into dynamic library `librinasim.so/dll` which contains both RINASim core and policies and allows to run simulations.

3.4. OMNeT Handbook

OMNeT is a discrete event simulator that is freely available for academic purposes. A page dedicated to the simulator and its community is [omnetpp-main]. It is a general simulator that is easily extensible because of its modular nature. Additional frameworks include:

- INET and ANSAINET - wired computer networks [omnetpp-inet] and [omnetpp-ansa]
- INETMANET and MIXIM - wireless and mobile computer networks [omnetpp-mixim]
- OverSim - peer-to-peer computer networks [omnetpp-oversim]
- Veins - traffic and mass transportation networks [omnetpp-veins]
- Castalia - wireless sensor networks [omnetpp-castalia]

A comprehensive OMNeT manual covering simulation core is available at [omnetpp-manual] or for people familiar with simulation is more suitable its quick-reference variant [omnetpp-ide].

3.4.1. Basics

OMNeT is using a hierarchical structure of simulation modules. Top level system modules consist of submodules or so-called compound modules that could be either further divided according to a child-parent scheme, or that are undividable and thus named simple modules.

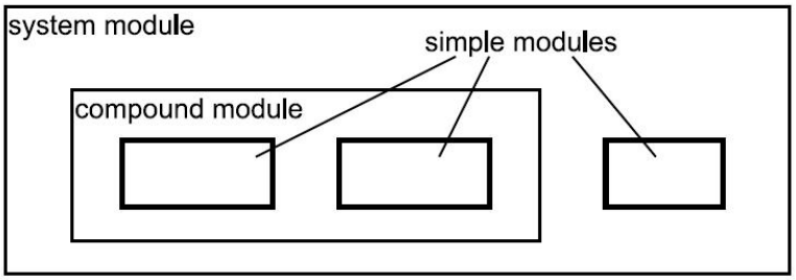

Figure 6. OMNeT module structure

OMNeT is the object-oriented simulator that leverages two languages: 1) NED for network topology description and modules interconnections; 2) C++ for simulation modules behavior. Modules communicate with each other by sending messages (either in the form of PDUs or timer notifications). Messages could be received either from neighbor modules or the same module (self-messages). A module may contain input (for receiving) and output (for sending) gates. Connections are created between gates. The connection can exist between sibling modules or modules with the parent-child relationship.

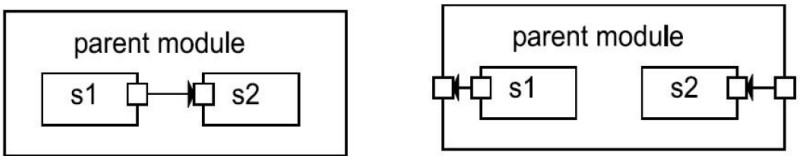

Figure 7. Parent/children modules

3.4.1.1. Simple modules

The NED language describes module´s structure (file with *.ned extension) and C++ implements its functionality (files with *.cc and *.h extensions).

```
simple TestModul
{
    parameters:
        @display("i=block/queue");
    gates:
        input in;
        output out;
}
```

Figure 8. Example of a simple module

Keyword simple defines module´s name TestModule where expected implementation should be in TestModule.cc and TestModule.h. The module contains two subsections - parameters and gates - where both are optional. In parameters section, different properties and variables (int, string, double, xml, etc.) are set. Parameters could be set on fixed value here, or dynamically in omnetpp.ini file that accompanies every simulation. Section gates consist of gates definitions (in the demo there are two gates, one input gate called in, and one output gate called out).

3.4.1.2. Compound modules

Compound modules aggregate multiple modules into a larger comprehensive unit.

```
module Router
{
    parameter:
        @display("i=block/router");
    gates:
        inout SerialInterface[];
        inout EthInterface[];
    submodules:
        tcp: TCP;
        ip: IP;
        layer1: physicalLayer;
    connections:
        tcp.ipIn <-- ip.tcpOut;
        tcp.ipOut --> ip.tcpIn;
        layer1.ipIn <-- ip.l1In;
        layer1.ipOut --> ip.l1Out;
}
```

Figure 9. Example of a compound module

The name of a compound module follows after the keyword "module" (in the example it is Router). Section parameters and gates have the same semantics as in the case of any simple module. Section submodules define references together with the name of imported submodules. Section connections define how input and output gates are bound together (for instance the IP layer gate named tcpOut is connected with TCP's ipIn). The output gate is marked as #-, the input as -$\rightarrow$ and bidirectional connections as $\leftarrow\rightarrow$.

3.4.1.3. Network modules

The highest level of abstraction is provided by network modules that describe the whole topology of a different compound and simple modules. Once again it is outlined in the NED language but with the different starting keyword "network".

```
network SimpleCircle
{
    submodules:
        router1: Router;
        router2: Router;
        router3: Router;
        router4: Router;
    connections:
        router1.interface++ <--> EthLink <--> router2.interface++;
        router2.interface++ <--> EthLink <--> router4.interface++;
        router3.interface++ <--> EthLink <--> router1.interface++;
        router4.interface++ <--> EthLink <--> router3.interface++;
}
```

Figure 10. Example of a network module

The previous snippet is an example of a simulation network with four routers interconnected in a ring topology.

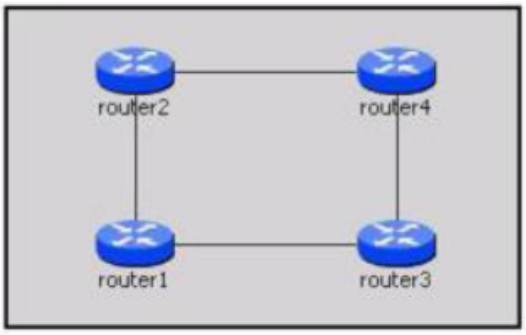

Figure 11. Four routers topology

3.4.2. Simulator and IDE

OMNeT uses the following component architecture:

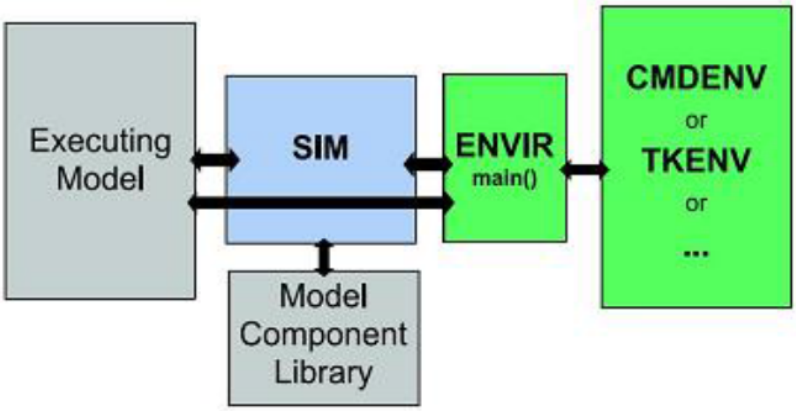

Figure 12. OMNeT component architecture

- Sim - Discrete event simulator core;
- Envir - Libraries shared by any user code consisting of event scheduler and dispatcher. Catches and handles exceptions;
- Cmdenv/Tkenv - Libraries for graphical or command line user interface. Allow interactive execution of simulations with step-by-step debugging and logging;
- Model Component Library - User implemented simulation modules;
- Executing Model - Compiled model of a given simulation scenario.

The OMNeT IDE is using Eclipse since version 4. A basic IDE introduction is available at [omnetpp-demo]. The most relevant keyboard shortcuts consist of:

- *Ctrl + B* = build (compile) simulation modules inside project;
- *Ctrl + F11* = run target simulation (either NED file or omnetpp.ini);

- *Ctrl + Tab* = switching between NED description and associated C++ source codes;
- *Alt + Left/Right Arrow* = switching between tabs;
- *Ctrl + Space* = Intelligent helper.

The following picture describes basic OMNeT++ IDE parts:

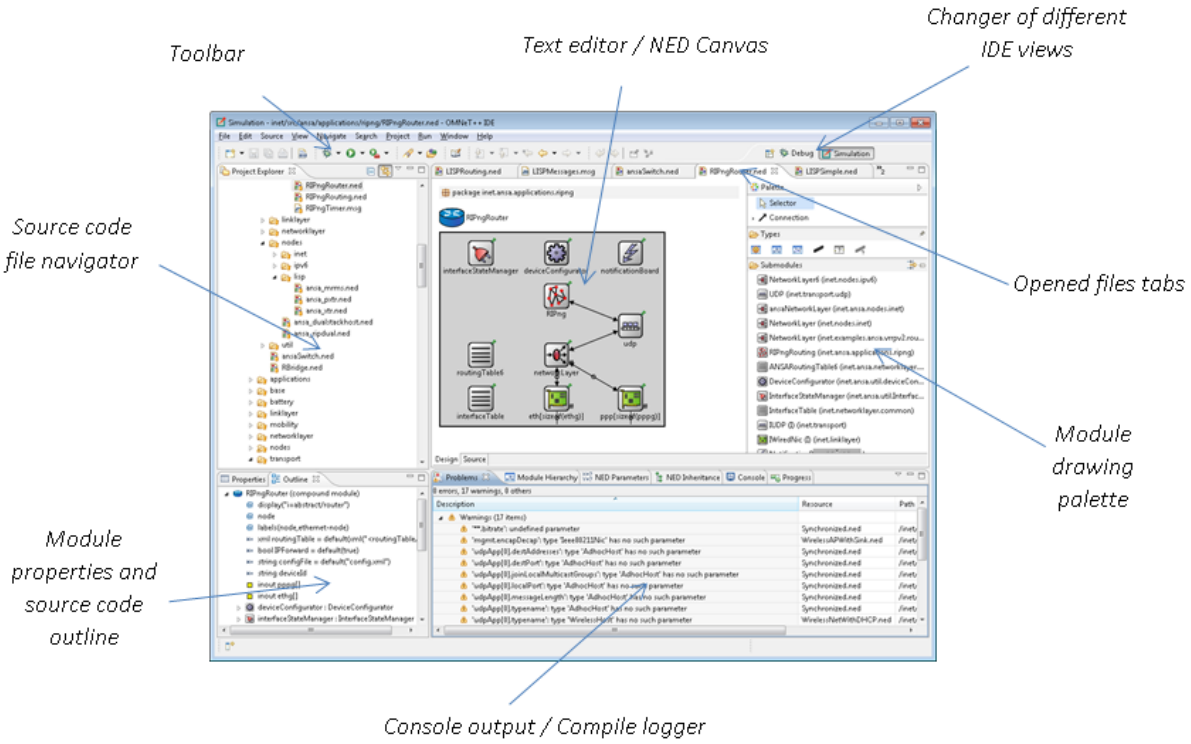

Figure 13. Basic OMNeT++ parts

Tcl/Tk environment starts after a simulation is successfully compiled and executed. The first window is for simulation visualization, and the second windows are for event logging:

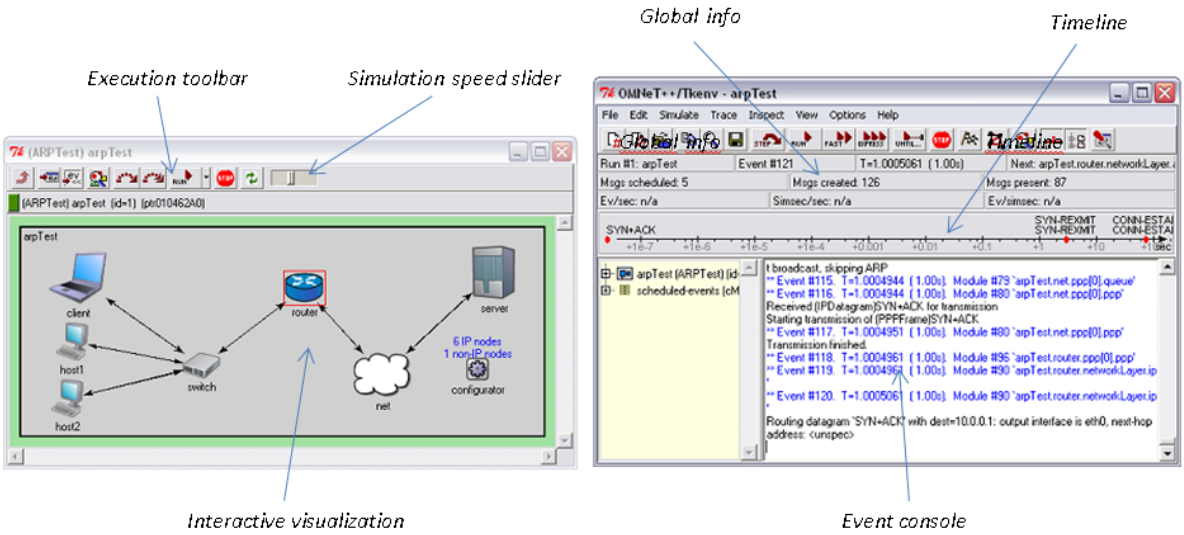

Figure 14. Event logging window

3.4.3. Tips and Tricks

This subsection contains few tricks that may come handy for any developer using RINASim.

3.4.3.1. Parallel build

Parallel build significantly increases the time of project compilation. OMNeT supports a parallel building of source codes since version 4.6 for any environment (including Windows platform). It is advised to check whether parallel build is enabled using _Project -> Properties_ windows and section _C/C Build_ tab *Behaviour* (see figure below)

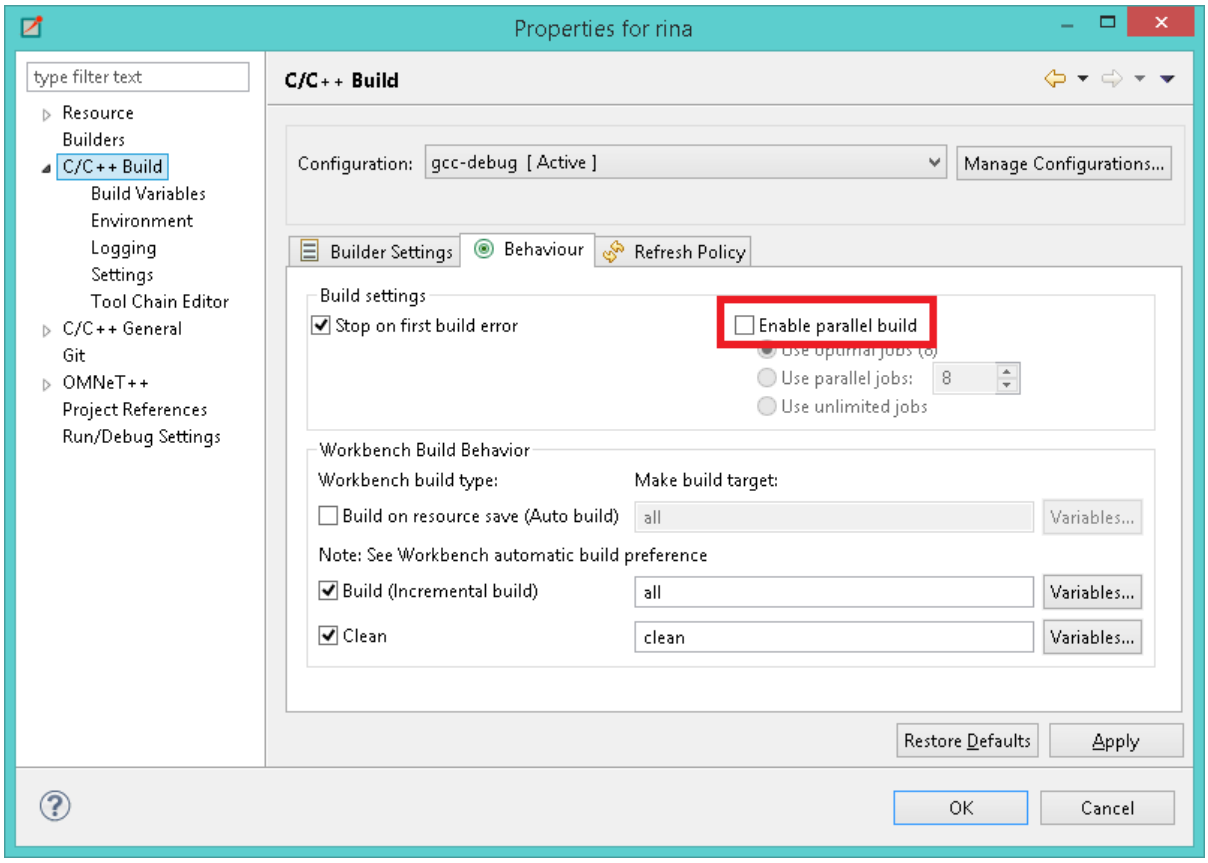

Figure 15. Enable parallel build through IDE

3.4.3.2. Visual aid

We created RINASim own code highlighter to get visual help during source navigation. It can be downloaded from this link [omnetpp-highlight]. Integration into IDE is straight-forward process. Just copy content of the file into *\samples\.metadata\.plugins\org.eclipse.core.runtime \.settings* inside your OMNeT++ installation directory. Illustration of highlighting is in the figure below.

Apart from that is recommended to use EditBox section highlighter, which is freely available (see [omnetpp-editbox]) Eclipse IDE plugin. It can be installed into OMNeT++ IDE using official Eclipse plugin manager, go to *Help # Install New Software*.

```
49  extern const char* PAR_FORCEORDER;
50  extern const char* PAR_MAXALLOWGAP;
51  extern const char* PAR_DELAY;
52  extern const char* PAR_JITTER;
53  extern const char* PAR_COSTTIME;
54  extern const char* PAR_COSTBITS;
55  extern const char* PAR_ATIME;
56
57
58  enum CDAPConnectionState {NIL,
59      FLOW_PENDING, CONNECTION_PENDING,
60      AUTHENTICATING, ESTABLISHED, RELEASING};
61
62  class AEBase : public cSimpleModule
63  {
64    public:
65      bool hasFlow(const Flow* flow);
66
67      const APNamingInfo& getApni() const;
68
69      bool operator== (const AEBase& other) {
72
73      const int getAuthType();
74      const std::string& getAuthName() const;
75      const std::string& getAuthPassword() const;
76      const std::string& getAuthOther() const;
77      void changeConStatus(CDAPConnectionState conState);
78      CDAPConnectionState getConStatus();
79      Flow* getFlowObject() const;
80      void setFlowObject(Flow* flowObject);
81
82    protected:
83      //Flows flows;
84      Flow* FlowObject;
85      APNamingInfo apni;
```

Figure 16. RINASim official source code highlighter

4. High-level design

To understand RINA architecture means to know each of its elements. This chapter starts with a description of high-level RINA network nodes and then goes deeper and outlines various IPC Management and IPCP components.

4.1. Nodes

There are only three basic kinds of nodes in RINA network (illustrated in the figure below). Each type represents 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.

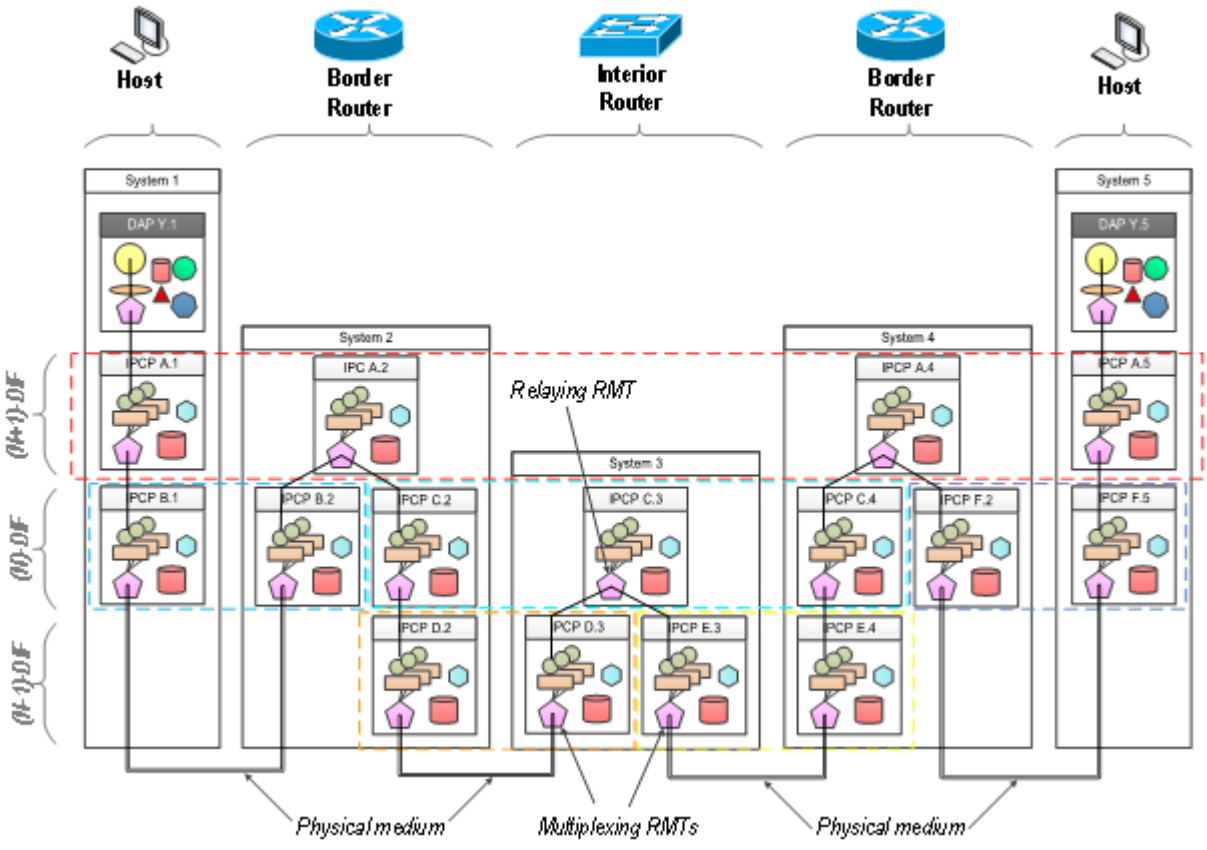

Figure 17. Example of RINA network with three levels of DIFs and different nodes

As seen in Figure above, 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, (N-2)-DIF is needed to reach physical medium.

4.2. DAF Design

IPC Management is an integral part of any DAP responsible for managing supporting DIFs and providing their services to participating APs. IPC Management consists of following components depicted in Figure below:

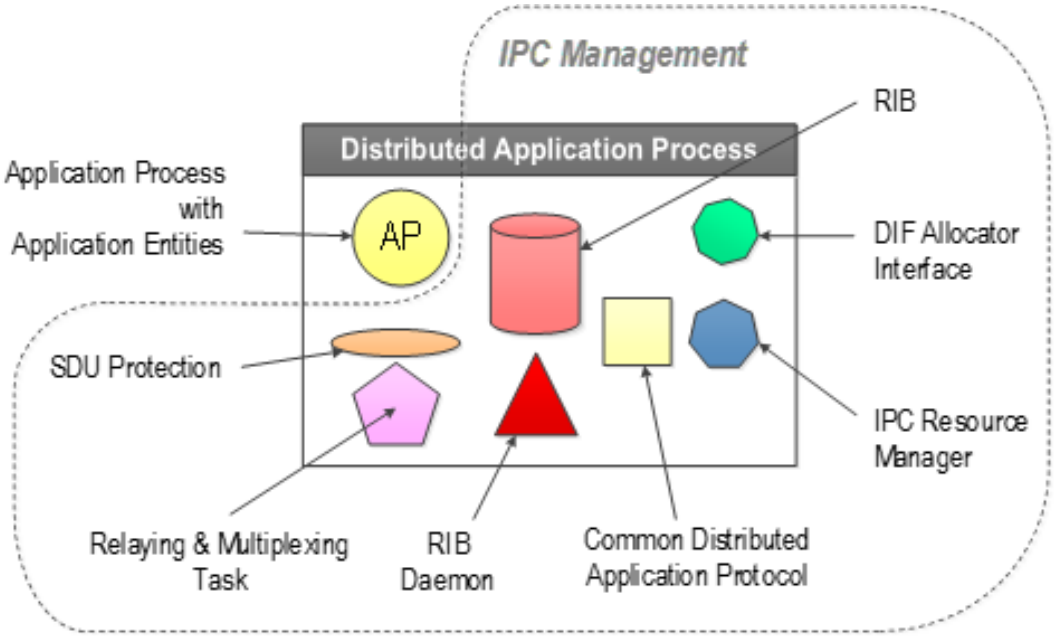

Figure 18. Distributed Application Process components

Only IPC Resource Manager and DIF Allocator interface are exclusive to IPC Management, other components are also present in IPC Process and described later.

4.2.1. DIF Allocator

The primary task of **DIF Allocator (DA)** is to return a list of DIFs where destination application may be found given ANI and access control information. Additional and more complex DA description is available in [RINA-layer-discovery]. DA contains and works with multiple mapping tables to provide its services:

- **Naming information table** – provides association between APN and its synonyms;

- **Search table** – provides mapping between requested APN and the list of DAs where to find 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.

4.2.2. IPC Resource Manager

IPC Resource Manager (IRM) (see specification [IRM-spec]) as its name suggests, it 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 manages the use of flows between AEs and DIFs. IRM may choose to multiplex a single or multiple AE flows into a 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).

4.3. DIF Design

IPC Process is instance within DIF, which allows the computing system to do IPC with other DIF members. Each IPC process performs (secure/reliable) data transport, (authenticated) enrollment, (de)allocation of resources, routing, management and more. Functions could be categorized under one of following categories: a) data transfer; b) data transfer control; and c) IPC management. Each category with different processing timescale and complexity – a) is simplest and performed the most often, c) the least often but the functionality is rather complex.

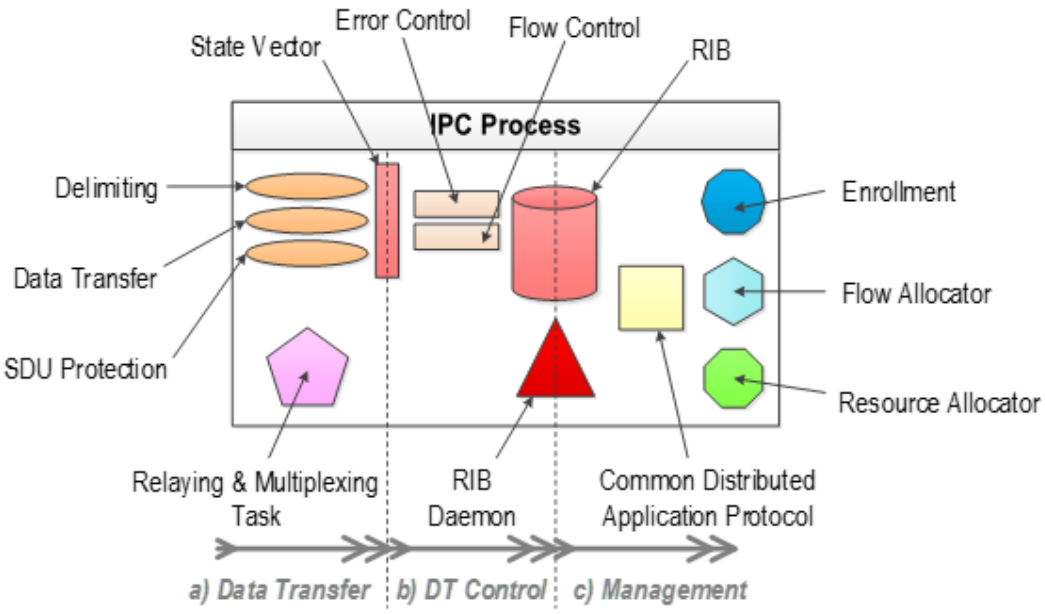

Figure 19. IPC Process components

IPC provides API to a DIF/DAF above, which requested its service. Basic **IPC API** offers four operations: *allocate* (allocates communication resources); *deallocate* (releases previously allocated resources); *send* (passes SDU to IPC) and *receive* (retrieves SDU from IPC). Calls may be further subdifferentiated as *allocate request*, *allocate response*, *deallocate submit* and *deallocate deliver*.

Graphical representation of IPC Process and its most important components is depicted in Figure above. A brief description of each component and their functionality is provided below figure. Some components outlined below also contain policy descriptions. Those policies are mentioned because they are relevant to our contribution.

4.3.1. Enrollment

Enrollment takes place whenever IPCP joins existing DIF. IPCP newcomer creates a connection with other IPCP (which is already a member) allocating (N-1)-flow. Enrollment occurs after successful connection establishment. Enrollment procedure of a new member should be dependent on a connection use-case. For instance, there may be a different exchange of messages for: a) the new member joining DIF for the first time; b) the IPCP that had been already a member of DIF and right now is rejoining. The new member either tells or gets its address to/from a DIF. Enrollment procedure is codified in [Enroll-spec].

Detail description of Enrollment operation is provided in Figures below. Transitions are denoted with “input / action” labels. There are two different FSMs. The first figure describes initiating process right after finished CACE Phase. The second figure shows responding process after

CACE Phase. Only correct transitions are shown. Either Initiating or Responding process can invoke deallocate in any state.

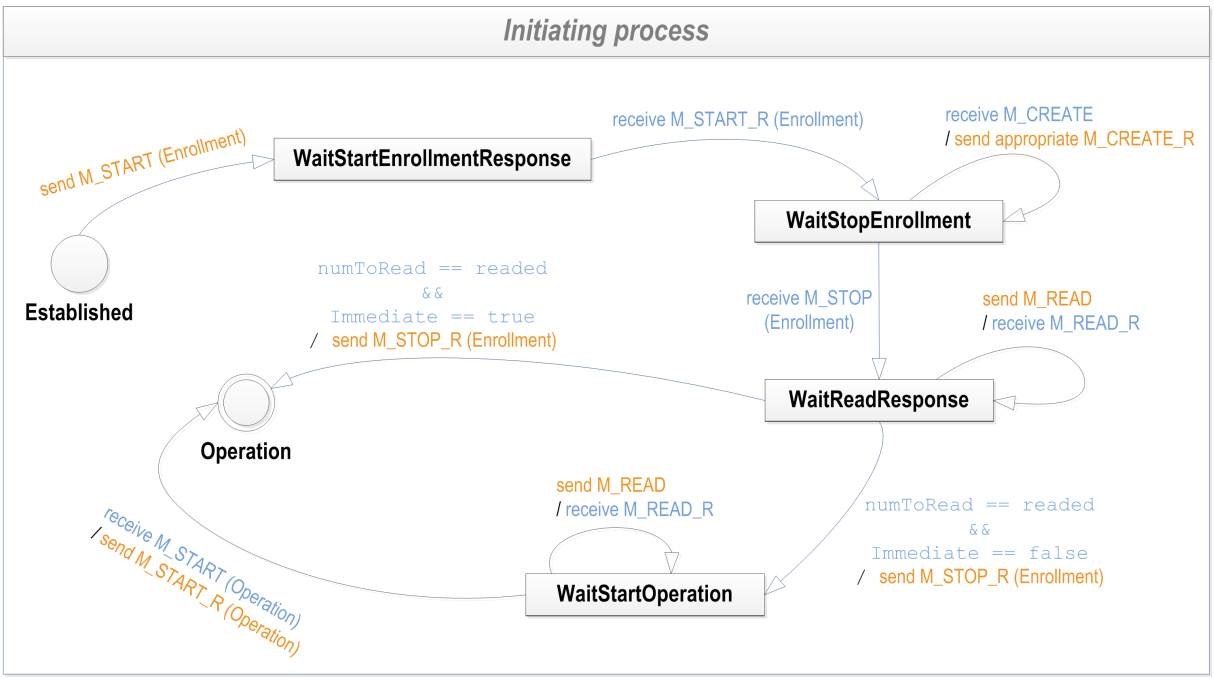

Figure 20. Initiating process Enrollment State Diagram

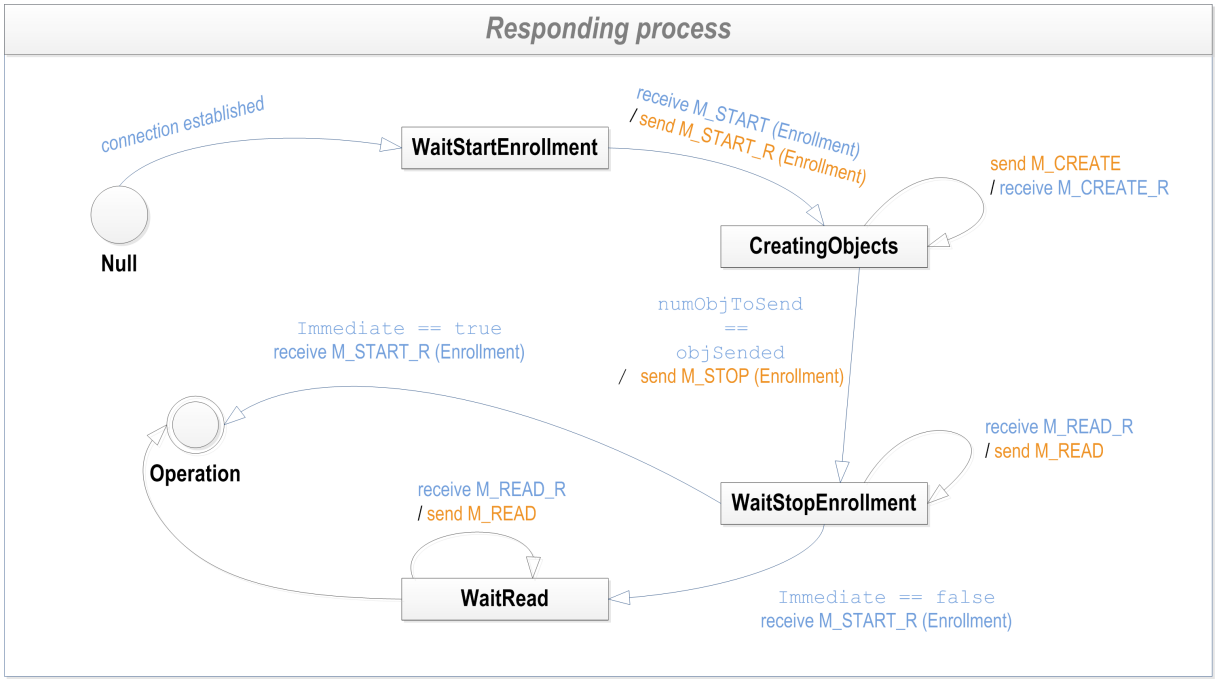

Figure 21. Responding process Enrollment State Diagram

Both processes are using CDAP messages for communication.

4.3.2. Delimiting

SDU in RINA is a contiguous chunk of data. IPC might fragment SDU (when passing it down) or combine user-data (when passing it up). Hence, the operation performed by **Delimiting** module (for specification see [Delim1] and [Delim2]) is to delimit SDU into/from 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 in PCI) delimiting could be used.

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

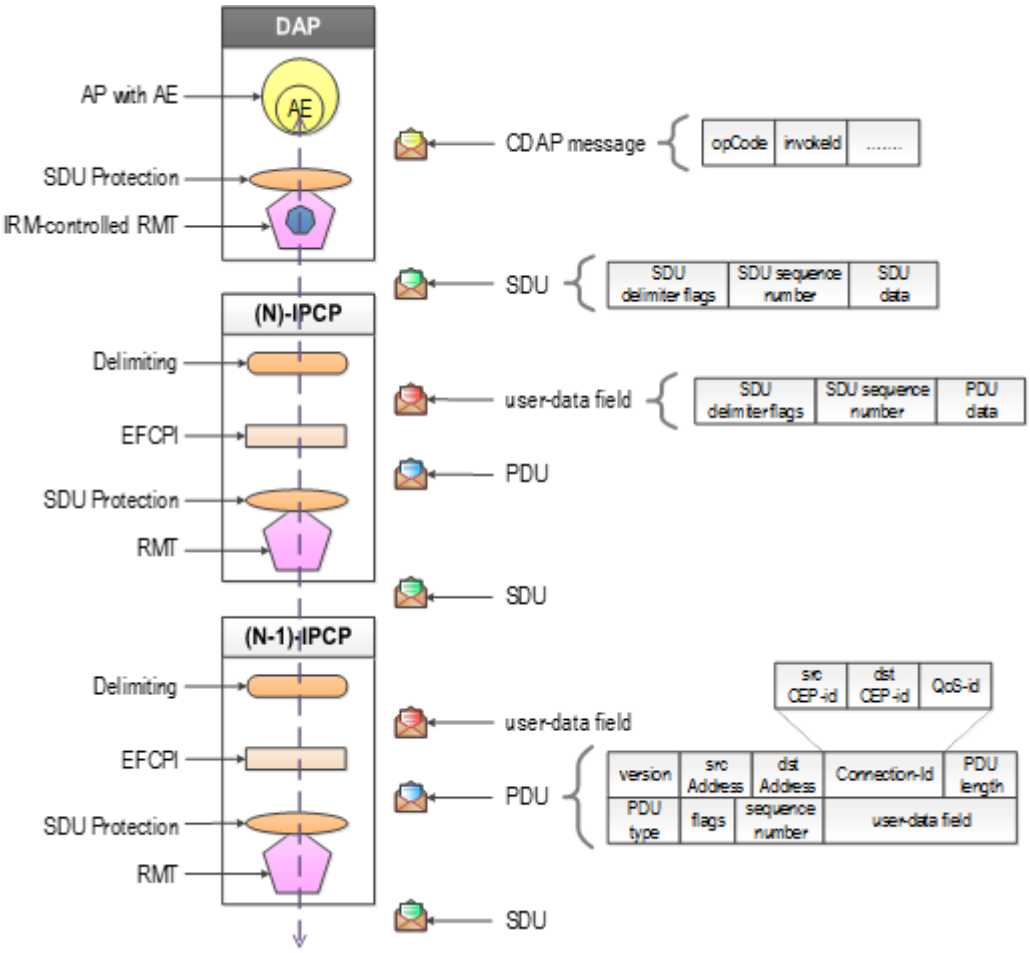

Figure 22. Message passing between RINA components

4.3.3. Data Transfer with Error and Flow Control

Error and Flow Control Protocol (EFCP) is split into two independent PMs coupled and coordinated through a state vector. As EFCP name suggests, EFCP guarantees data transfer and data control. Full EFCP functionality is described in [EFCP-spec]. However, these specifications are currently being revisited.

Data Transfer Protocol (DTP) implements mechanisms tightly coupled with transported SDUs, e.g., reassembly, sequencing. DTP PM operates on a data PDU's PCI with fields requiring minimal processing – source/destination addresses, QoS requirements, Connection-id, optionally sequence number or checksum. DTP carries user-data.

Data Transfer Control Protocol (DTCP) implements mechanisms that are loosely coupled with transported SDUs, e.g., (re)transmission control using various acknowledgment schemes and flow control with data-rate limiting. DTCP functionality is based on Watson's Delta-t and DTCP PM processes control PDUs. DTCP provides error and flow control over user-data.

There is **EFCP instance (EFCPI)** module per every active flow. EFCPI consists of DTP and DTCP submodules. DTCP policies are driven by the quality of service demands. 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 figure below. Depicted are also data transfer and data control transfer paths. Control traffic stays out of the main data transfer.

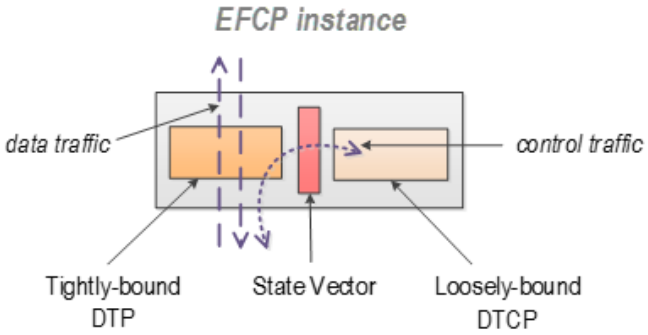

Figure 23. EFCP instance divided into DTP and DTCP part

4.3.4. Relaying and Multiplexing

Relaying and Multiplexing Task (RMT) modules have two main responsibilities – relaying and multiplexing as characterized in [RMT-spec]. The goal of multiplexing is to pass PDUs from EFCPIs and RIB Daemon 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 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 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.

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 observed IPC process performance.

Each IPC process has to solve the forwarding problem: given a set of EFCP PDUs and (N-1)-flows leading to various destinations, to which flow should be each PDU forwarded? In RINA, the decision is handled by the RMT and its *PDUForwardingPolicy*. The *PDUForwardingPolicy* may consist of looking up the PDU's destination in its forwarding table (resembling the forwarding mechanism in traditional TCP/IP routers), but it is not a requirement; other experimental forwarding paradigms (such as forwarding based on topological addressing) may not require a forwarding table at all. When in need of deciding for an output (N-1)-port for a PDU, the *PDUForwardingPolicy* is given the PDU's PCI and then it returns a set of (N-1)-ports to which the PDU has to be sent. This provides enough granularity to implement multiple communication schemes apart from unicast (such as multicast or load-balancing) because the decision is left to the *PDUForwardingPolicy*. E.g., a simple forwarding policy would return a single (N-1)-port based on PDU's destination address and QoS-id, whereas in case of a load-spreading policy and multiple (N-1)-ports leading to the same destination, the policy could split traffic by PDUs' flow-ids and always return a single (N-1)-port from the set.

4.3.5. SDU Protection

SDU Protection is the last part of the IPC Process 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. It also provides mechanisms for data compression or other two-way manipulations that depend on the (N-1)-flow used and can increase the effectiveness of other SDU Protection mechanisms.

All the mechanisms provided by the SDU Protection module are encapsulated in two primary functions: **protect_sdu** and **unprotect_sdu**. These functions are called by the RMT, connecting the SDU Protection module to the rest of the IPC Process components. The **protect_sdu** function is called after the RMT decides which (N-1)-port will the SDU be passed to whereas the **unprotect_sdu** function is the first function called after receiving data from an (N-1)-port.

Due to different levels of trust an (N)-DIF can have towards different (N-1)-DIFs, SDU Protection handles each (N-1)-flow on it's 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 could look like the following:

- **Null SDU Protection** policy that performs no transformations

- **Basic SDU Protection** which applies lifetime limiting (TTL) and error checking (CRC)
- **Cryptographic SDU Protection** which extends the Basic policy by adding cryptographic encryption of data and an integrity check using a cryptographic hash of the content

4.3.6. Flow Allocator

Flow Allocator (FA) 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 controller/container for all Flow Allocator Instances.

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 maintaining a single flow's connection. FAI returns port-id to the allocation requestor upon satisfactory allocation as a referencing handle. FAI participates only on port allocation, not on synchronization, which is the responsibility of 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 DIF. This activity involves maintaining the table with entries that map requested ANI to IPCP's address.

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. 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*.

Flow allocation processes for (N)-DIF between two APs on different systems is depicted in the Figure below. It assumes that relevant (N-1)-flows have been already allocated using the same principle as the one being described but on different DIF's rank.

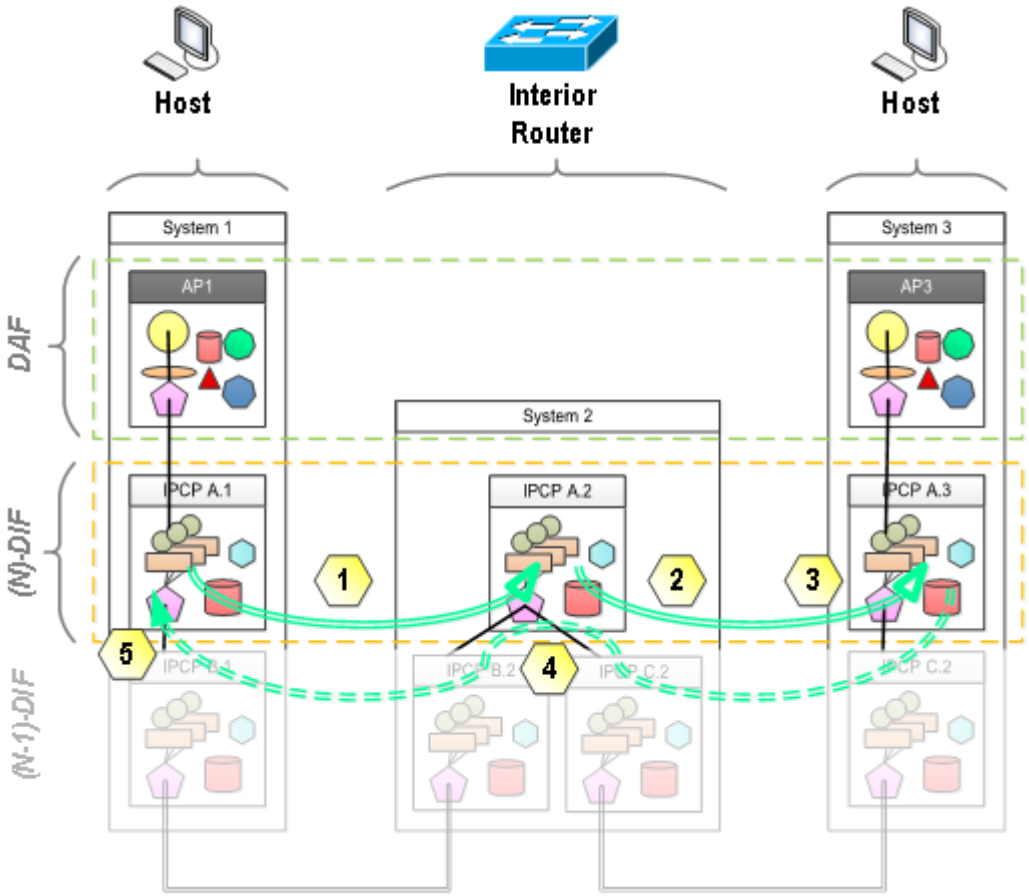

Figure 24. Flow allocation process

- AP1 issues *allocate request* that is delivered to IPCP A.1. If it is valid and well-formed, then it spawns FAI to manage requested flow. FAI resolves AP3's APN to one of DIF A addresses (A.3). It instantiates EFCPI (with CEP-id) and creates bindings between EFCPI and RMT. *Create flow request* is sent as the last step;
- *Create flow request* arrives at "System 2". IPCP A.2's FA processes the request and discuss NMS. It discovers that request is not intended for any local AP. FA looks up the destination discovering that A.3 should be a next-hop. FA forwards the request to "System 3";
- The request arrives at IPCP A.3. Over there, FA determines by querying NMS that *create flow request* destination address is its address. Thus, destination AP resides on this system. FAI is spawned and determine whether the request can be accommodated. If not then negative *create flow response* is sent back to the requestor. Otherwise, FAI notifies destination AP with *allocate request*;
- If destination AP accepts or rejects the request then either positive, or negative allocate response is returned to FAI. Based on the response, FAI binds port-id, instantiates EFCPI, creates bindings. Flow object is updated (with local port-id and CEP-id) and sent back as positive/negative *create flow response*. Response is just relayed (not processed) on interior routers (IPCP A.2);

- Originating A.1's FAI receives *create flow response* and updates relevant flow object. If the response is positive, then, FAI notifies source AP with positive *allocate response* and APs may commence data transfer. If the response is negative, then FAI invokes retry policy to correct flow creation or deal appropriately with failure (i.e., passing negative *allocate response*).

Original specification [FA-spec] were refined as the subject of this thesis contribution. Detail description of flow allocation and deallocation is provided in Figures below. Transitions are denoted with "input / action" labels. FA and FAI maintain state for any given flow and refuse inappropriate transitions (e.g., initiating deallocation before the allocation is successful). These transitions are omitted for clarity. There are four different FSMs. The first figure depicts FA operation reacting upon notification from RIBd. Second and third figures show flow allocation procedure for initiating and responding FAIs. The last figure illustrates flow's lifecycle after successful allocation, and it is mutual for both initiating and responding FAIs.

NewFlowRequstPolicy 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. *AllocateRetryPolicy* occurs whenever initiating FAI receives negative create flow response. This policy allows FAI to reformulate the request and/or to recover properly from failure. *AllocateNotifyPolicy* controls a proper time when source AP is going to be notified of the result of allocation by initiating FAI. It may be either when EFCPI is created, or when allocation is confirmed by destination or any other notification strategy may be employed. *SeqRollOverPolicy* is invoked simultaneously by both initiating and responding FAIs whenever PDU's sequence number threshold is reached. The policy usually spawns new EFCPIs and changes bindings.

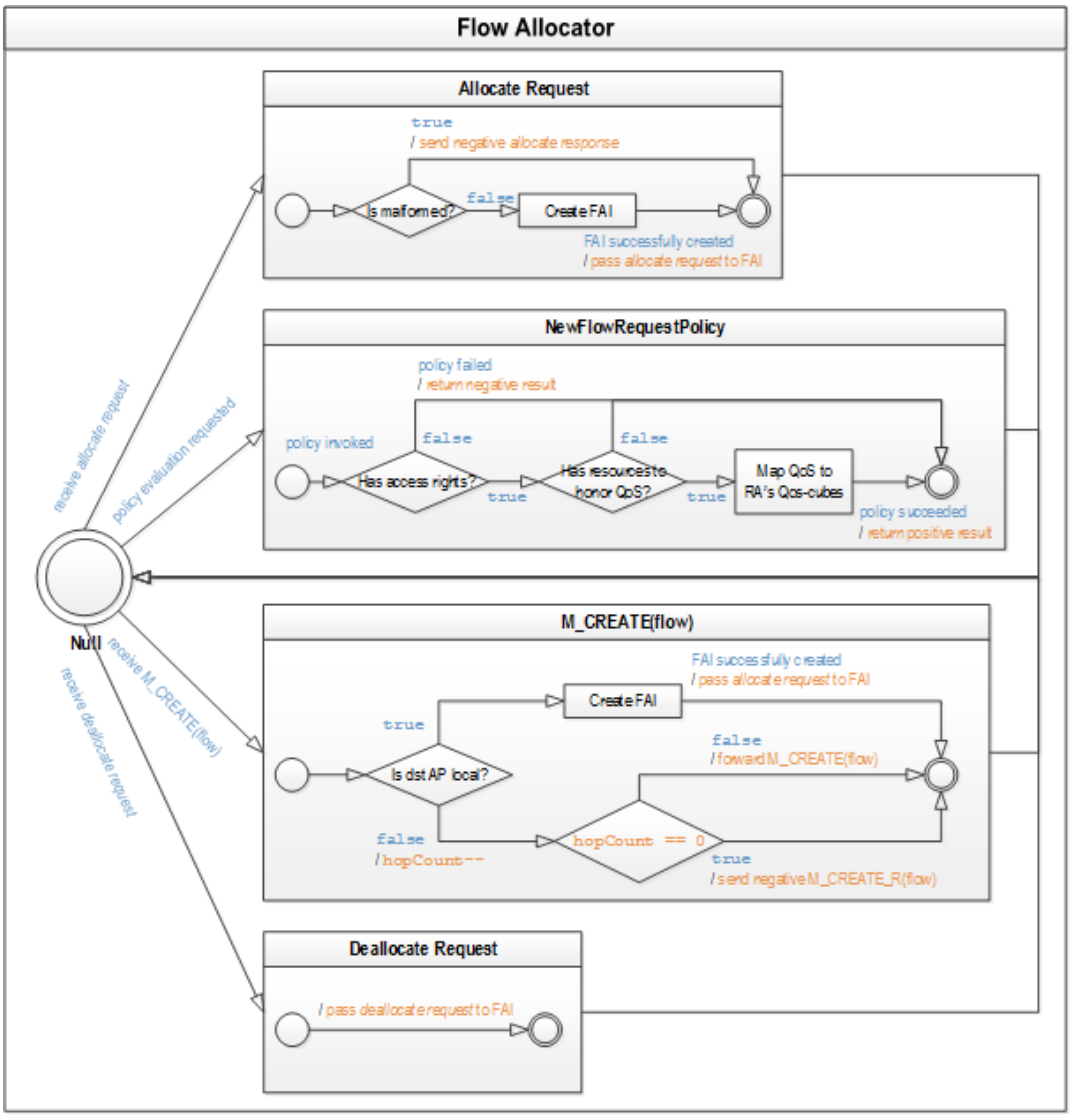

Figure 25. Flow Allocator operation

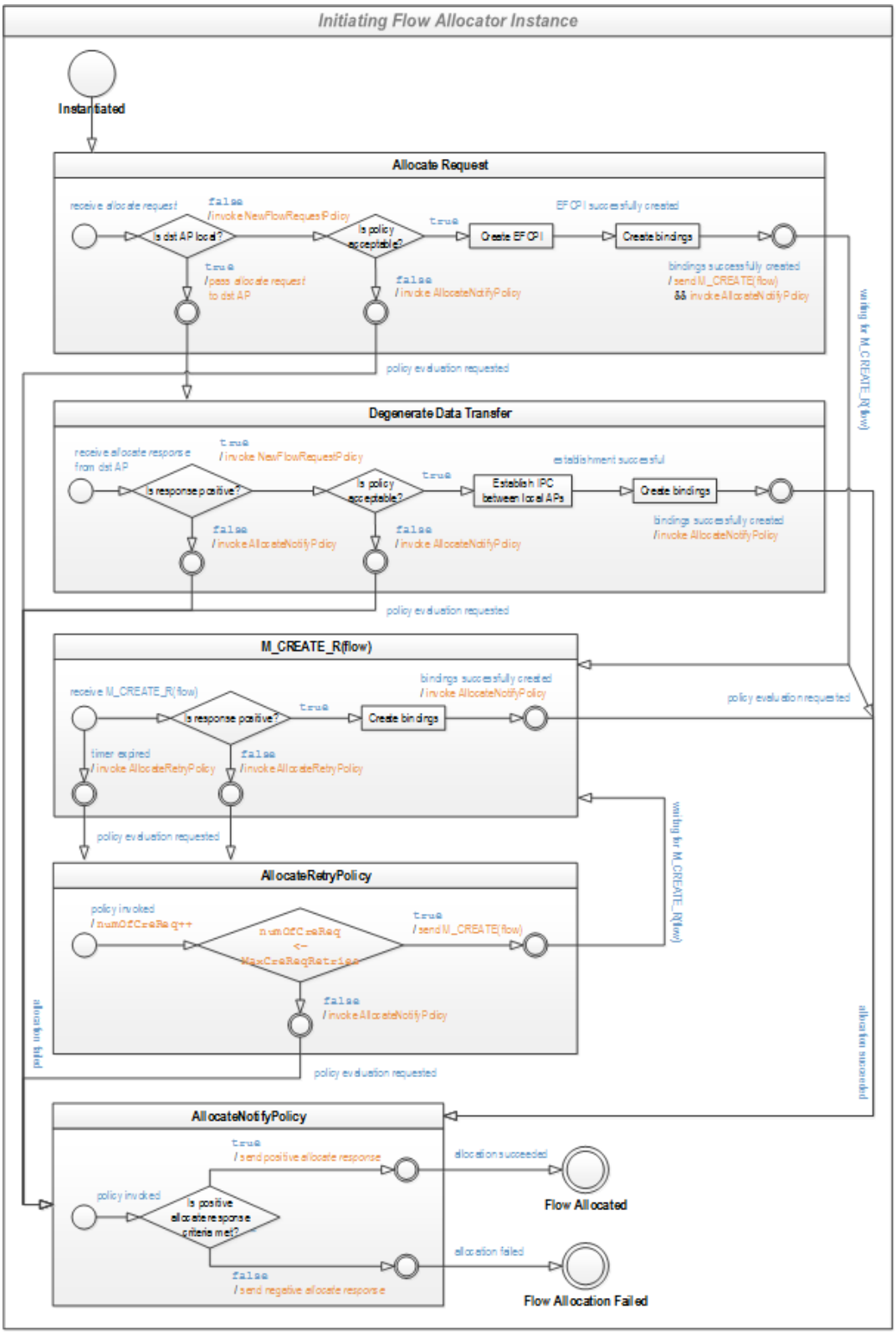

Figure 26. Flow Allocator Instance operation of initiating IPCP

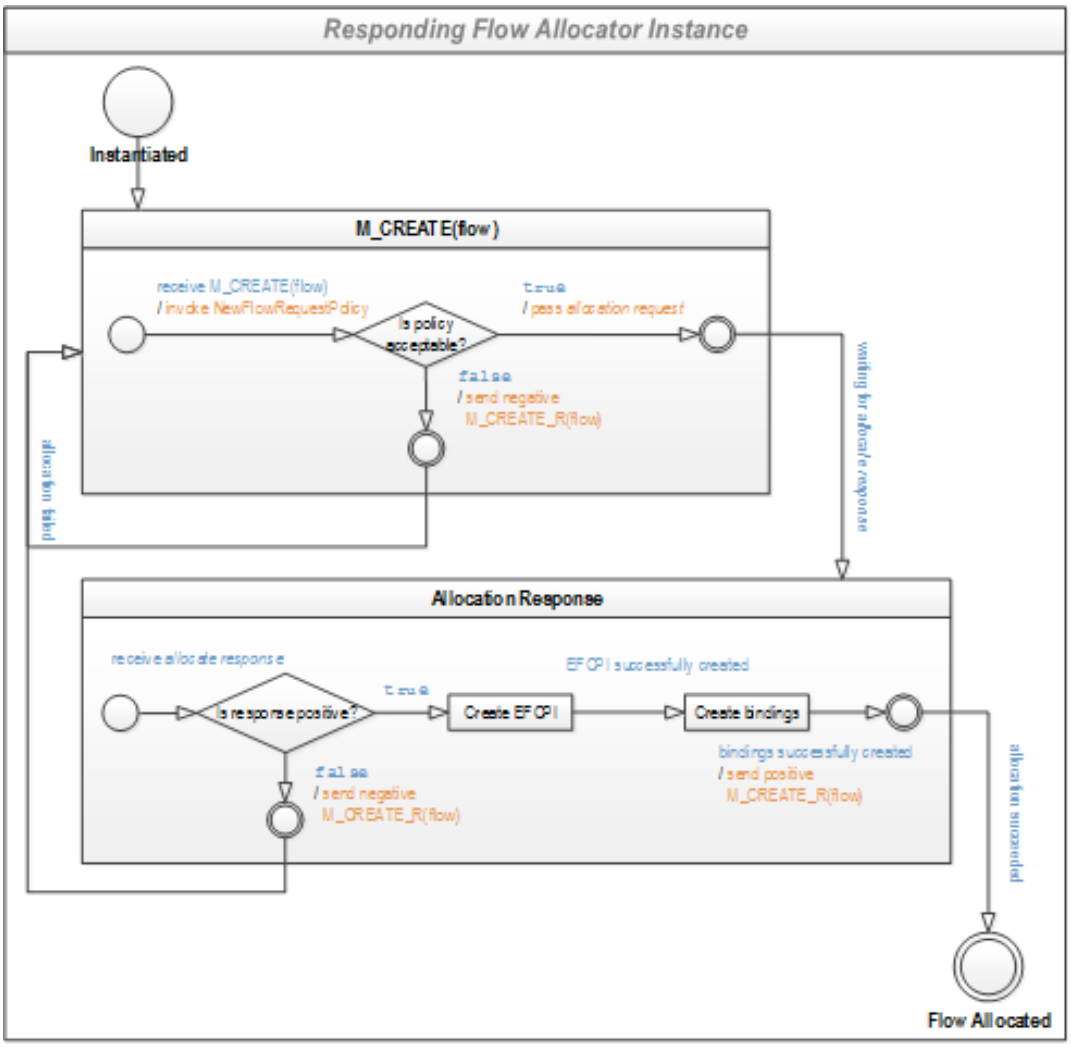

Figure 27. Flow Allocator Instance operation of responding IPCP before the flow was allocated

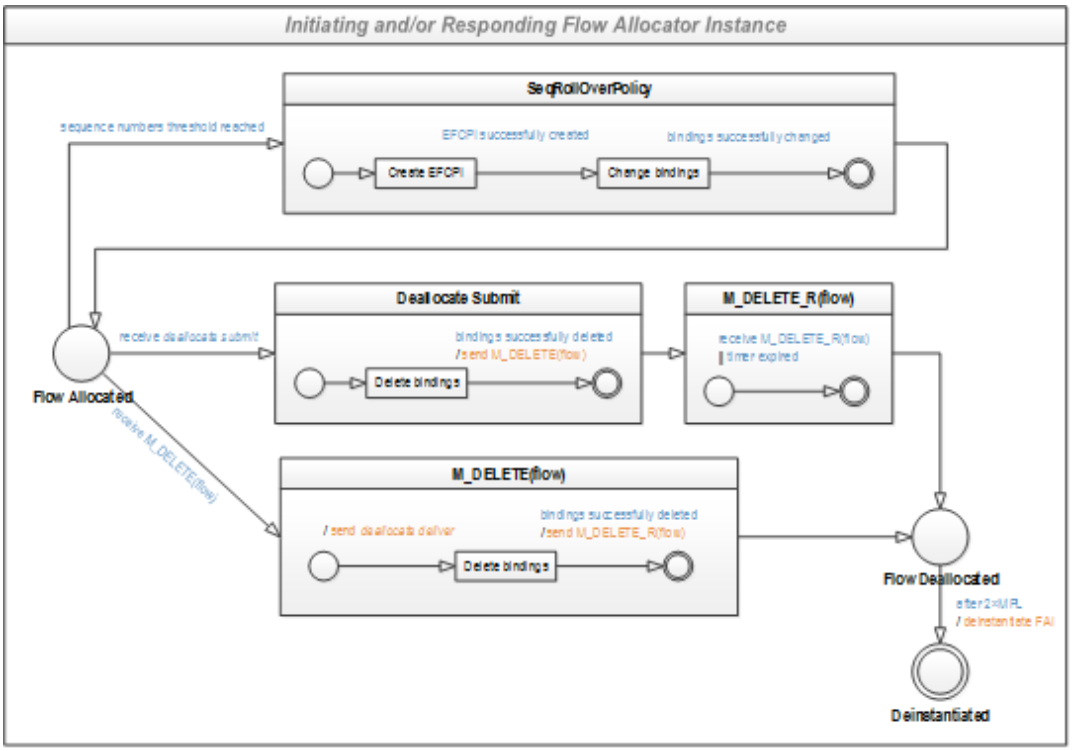

Figure 28. Flow Allocator Instance operation after the flow was allocated

4.3.7. Resource Allocator

If a DIF has to support different qualities of service, then various flows will have to be allocated to different policies and traffic for them treated differently. **Resource Allocator (RA)** delineated in [RA-notes] is a component accomplishing this goal by handling 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);
- maintains 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 DIF.

Generating information necessary for *PDUForwardingPolicy* is one of the tasks of RA, namely its subcomponent called **PDU Forwarding Table Generator**. For this purpose, RA uses pieces of information provided by other sources, most notably the *RoutingPolicy*.

The *RoutingPolicy* exchanges information with other IPCPs in the DIF in order to generate a next-hop table for each PDU (usually based on the destination address and the id of the

QoS class the PDU belongs to). The next-hop table is then converted into a **PDU Forwarding Table** with input from the PDU Forwarding Table Generator, by selecting an N-1 flow for each "next-hop". *RoutingPolicy* may resemble distance vector and link-state routing protocols used in today's Internet, but the current research is also aimed at other paradigms such as topological/ hierarchical routing, greedy routing or MANET-like routing.

4.3.8. RIB Daemon

All information maintained by IPC tasks such as FA, RA, and others is available and updated through **RIB Daemon (RIBd)** described in [mobj-spec] and [RIB-notes]. Information exchange is necessary to coordinate the distributed IPC. Different update strategies for various types of information may be used to synchronize state between different DIF member subsets.

Resource Information Base (RIB) is a logical database of information accessible via RIB Daemon. By logical database, we mean that some of RIB information may be stored in the dedicated database and the rest of IPCP components. Periodic or solicited events can cause RIB to be queried/updated by IPCP peers via management CDAP messages. RIBd provides an API to perform an operation on both local and remote RIB.

4.3.9. Common Distributed Application Protocol

RINA principles postulate that there is only a single application protocol required and this is the **Common Distributed Application Protocol (CDAP)**. DIFs use CDAP for all non-data communication (i.e., IPC management such as maintaining RIB, controlling flow allocation, joining a DIF). DAFs may not use CDAP for backward compatibility. However, CDAP expressiveness should allow the transition of legacy protocols. CDAP is based and patterned on two existing protocols – ACSE (see [isoiec-15953] and [isoiec-10035-1]) for the establishment phase, CMIP [isoiec-9596-1] for the data transfer phase.

CDAP subpart for data transfer is object-oriented (with built-in scope and filter support) protocol offering six primitive operations: *create*; *delete*; *read* (i.e., get value); *write)* (i.e., put or set value); *start* (i.e., execute action) and *stop* (i.e., suspend action). The collection of objects is dependent on used AE, which provides access rights to them.

CDAP has modular structure composed of three submodules to provide flexibility:

- The **common application connection establishment (CACE)** submodule;
- The **authentication (Auth)** submodule provides authentication of the communication endpoints. A range of submodules will be available to support different kinds (e.g., none authentication, shared password, certificates) of authentication policies employing various cryptographic tools (e.g., a-/symmetric ciphers for confidentiality, MAC codes for integrity);

- The CDAP submodule.

CDAP offers following eighteen message types summarized in Table below [CDAP]:

Opcode	**Description**
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 close operation
M_CREATE	Create an application object
M_CREATE_R	Response to M_CREATE carries result of creating request, including identification of the created object
M_DELETE	Delete a specified application object
M_DELETE_R	Response to M_DELETE, carries result of 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 value 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 cancelation
M_WRITE	Write a specified value to a specified application object
M_WRITE_R	Response to M_WRITE carries result of write operation
M_START	Start the operation of a specified application object, used when the object has operational and non-operational states

Opcode	Description
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

Connection management between two applications is divided into two traditional phases – establishment and data transfer. An AP issues *allocate request* to underlying DIF's IPCPC specifying the destination APN and QoS requirements. If the allocation is successful, IPCP returns port-id to be used as a handle for all communication leveraging this flow. When the previous phase is completed, CACE sends a *M_CONNECT* message to start authentication using Auth submodule. Additional message exchange might follow in order to support different authentication mechanisms. If it is successful then the connection is established and CDAP transits to data transfer phase.

Another contribution is further refinement of CACE specifications [CACEP]. Detail description of CDAP operation is provided in Figures below. Once again transitions are denoted with "input / action" labels. There are three different FSMs. The first figure depicts establishment phase on initiating the process. The second figure shows the same but from the perspective of the responding process. The third figure outlines data transfer phase for both initiator and responder once they successfully reach "Established". For the sake of readability, only correct transitions are shown. Incorrect transitions upon receiving unexpected CDAP message terminate from any state in "Error" marked as "wrong input". Both initiator and responder might "indicate deallocation", thus entering "Deallocating" state at any given moment.

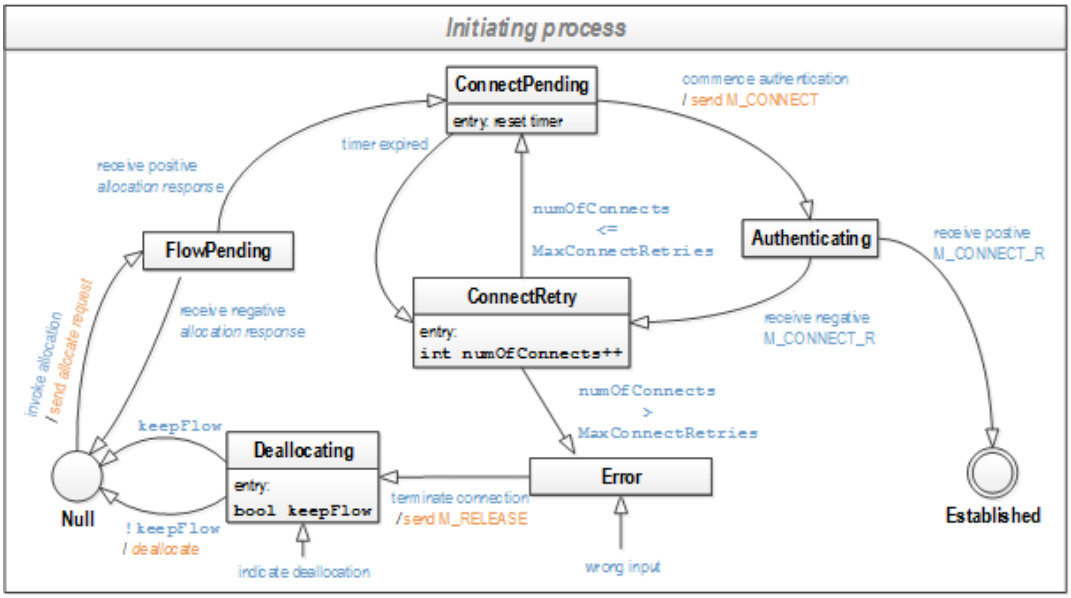

Figure 29. Establishment phase on initiating process

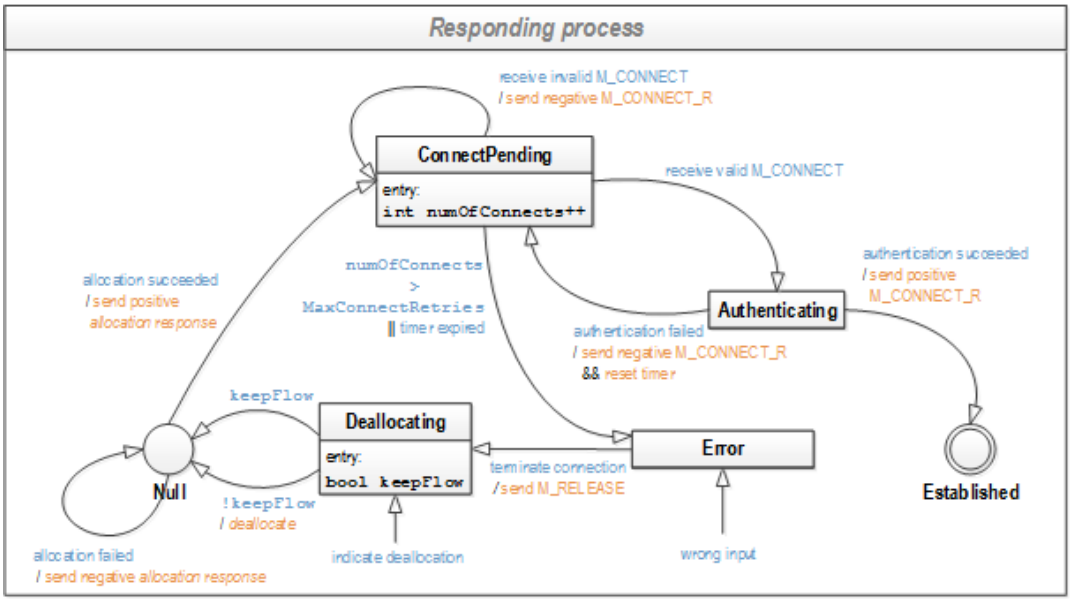

Figure 30. Establishment phase on responding process

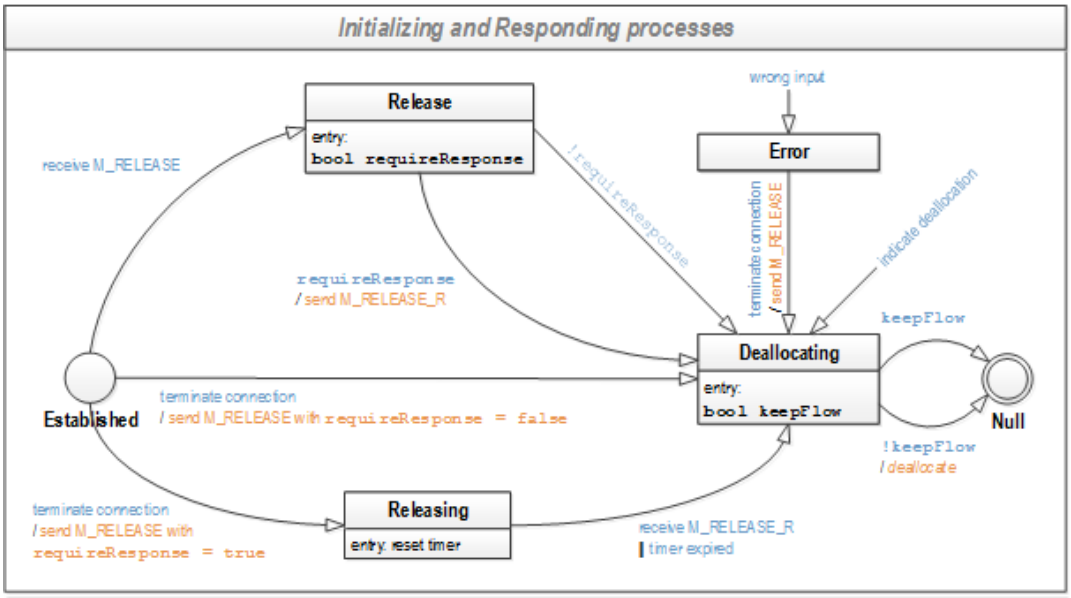

Figure 31. Data transfer phase on initiating/responding process

Depending on whether (N-1)-flow should be preserved or not, the transition from "Deallocating" (based on keepFlow boolean) may delete any state associated with connection and transit to the "Null" state.

4.4. Policy Framework

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. Thus, it is desired to design RINASim in a way that allows for the definition of policies and their smooth integration in the simulation models.

Hence, RINASim provides support for user-modifiable policies specifying the behavior of miscellaneous parts of RINA stack functionality. The separation of mechanism and policy is achieved by splitting the policy procedures into their separate modules — i.e. each policy invocation is done by calling an appropriate method of the proper policy's module.

An overview of available policies and policy implementation can be found in Sections 6 & 7.

4.4.1. Description

To minimize the need for modifying existing C++/NED source codes, the RINASim policy framework is based on OMNeT++ NED module interfaces. Each policy inside the DAF & DIF architectures is represented by a placeholder interface and the type of desired policy implementation is then determined at the simulation setup phase by a parameter placed in an

INI config file. This allows for virtually unlimited amount of user policy implementations to be defined and easily switchable via the configuration files.

In the default setting, each policy of each submodule uses its default policy implementation specified in the encompassing submodule's NED file (this default policy is usually a no-op placeholder). E.g., the default policies used by the Relaying and Multiplexing task are visible in /src/DIF/RMT/RelayAndMux.ned:

```
string schedPolicyName    = default("LongestQFirst");
string qMonitorPolicyName = default("SimpleMonitor");
string maxQPolicyName     = default("TailDrop");
```

Default policies loaded by the simulation:

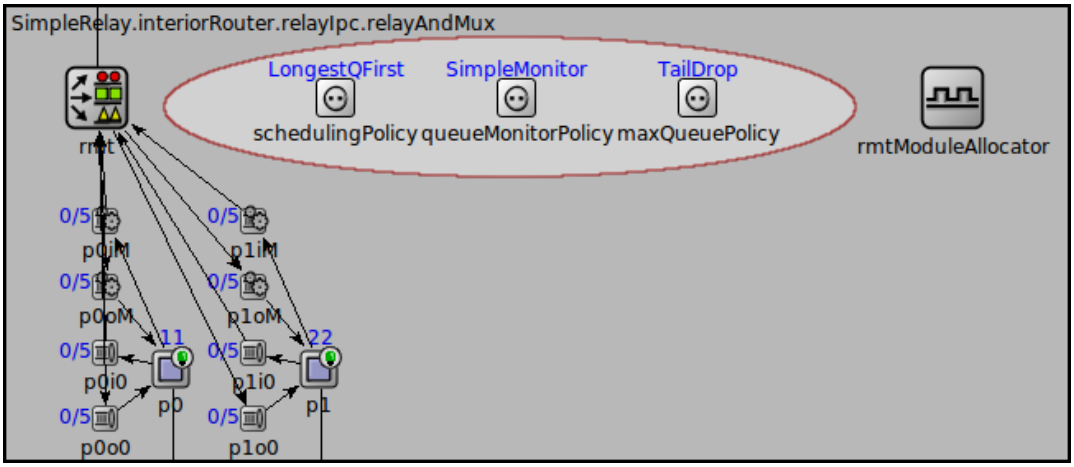

Figure 32. Default policy settings

4.4.2. Using the policy framework

Each policy consists of a NED module interface (e.g. "policies/DIF/RA/QueueAlloc/IntQueueAlloc.ned") and a C++ implementation interface (e.g. "policies/DIF/RA/QueueAlloc/QueueAllocBase.{cc,h}").

In case of creating a new policy implementation, the policy writer has to

- create a new simple NED module implementing the policy's interface, and
- implement this module by creating a new C++ class inheriting from the base C++ class and redefining desirable methods.

A new policy implementation can be loaded by setting a proper parameter of the encompassing module in the configuration file (e.g.

"host.ipcProcess0.resourceAllocator.queueAllocPolicyName = "QueuePerNFlow""). The parameter value has to match the name of the NED policy implementation module. Otherwise, the simulation framework will issue a fatal error in the initialization phase of the simulation.

4.4.3. Example usage

4.4.3.1. Use case

A user is working with the simulation scenario SimpleRelay[PingFC] which presents an example of two hosts communicating through an interior router that is prone to congestion due to queuing delay.

<pic>

The user wishes to modify the simulation scenario configuration so that the top IPC process of the interior router uses RED queuing discipline, by which some of the PDUs get dropped early to prevent congestion.

4.4.3.2. Solution

The first step consists of implementing the policy. In this case, the policy implementations needed for simulating the RED algorithm are already available in RINASim:

- REDMonitor (/policies/DIF/RMT/Monitor/REDMonitor), an implementation of QMonitorPolicy
- REDDropper (/policies/DIF/RMT/MaxQueue/REDDropper), an implementation of MaxQPolicy

When the policy implementations are ready, we need to reconfigure the default settings in omnetpp.ini so the simulation uses them instead of the default ones.

```
**.interiorRouter.relayIpc.relayAndMux.maxQPolicyName = "REDDropper"
**.interiorRouter.relayIpc.relayAndMux.qMonitorPolicyName = "REDMonitor"
```

Note: The OMNeT++ IDE makes the parameter specification easier thanks to its auto-assist feature (Ctrl + Space lists all available policy implementations).

Now, when the reconfigured simulation is run, it uses the specified RED policies:

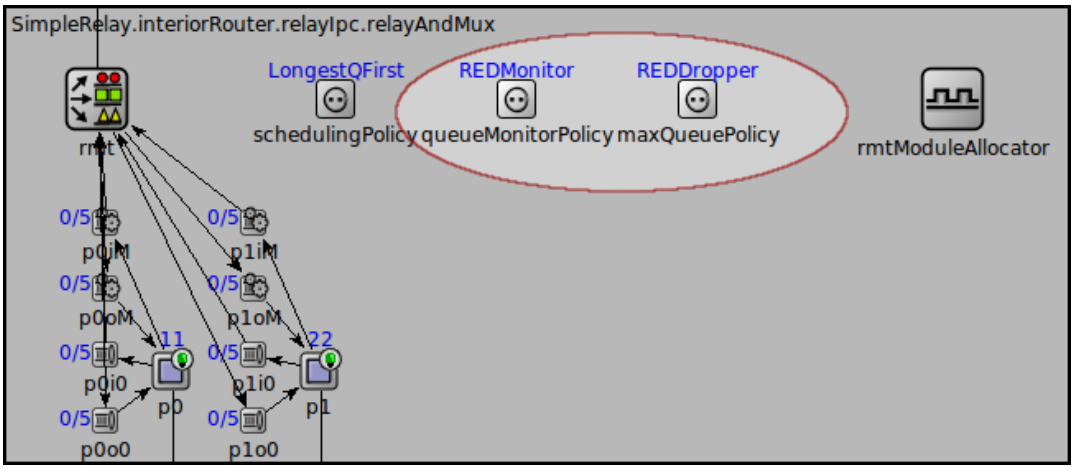

Figure 33. Overridden policy settings

4.5. Results Analysis

RINASim can record a detailed log about your message exchanges and collect various parameters values during the simulation run. This section outlines two features that are actually being implemented and used inside RINASim to gather data for research.

4.5.1. Collecting Statistics

OMNeT++ inherently supports signal-based statistic collection, see [omnetpp-stats] on which this subsection is loosely-based.

Signals are used to expose variables for result collection without telling where, how, and whether to record them. With this approach, modules only publish the variables, and the actual result recording takes place in listeners. Listeners may be added by the simulation framework (based on the configuration), or by other modules (for example by dedicated result collection modules).

The general guideline when creating a new placeholder for statistic analysis is:

1. Add `@statistic` properties to the simple module's NED file. A `@statistic` property defines the name of the statistic, which signal(s) are used as input, what processing steps are to be applied to them (e.g. smoothing, filtering, summing, differential quotient), and what properties are to be recorded (minimum, maximum, average, etc.) and in which form (vector, scalar, histogram). Accompany statistic declaration with source signal definition, but beware that statistic signal MUST NOT contain hyphen character in their name.
2. Later run the simulation and generate files with results (**.sca*, **.vci*, **.vec* and **.anf*). Inspect these results double-clicking on **.anf* file, which will open OMNeT++ build in results analyzer.

General OMNeT++ statistic definition examples:

```
@signal[qlen](type=int); // optional
@statistic[queueLength](source=qlen; record=max,timeavg,vector?);

@statistic[dropCount](source=count(drop); record=last,vector?);

@statistic[droppedBytes](source=sum(8*packetBits(pkdrop));
 record=last,vector?);
```

RINASim statistic definition example based on */src/DAF/IRM/IRM.ned* file:

```
@signal[IRM_PassUp](type=bool);
@signal[IRM_PassDown](type=bool);
@statistic[irm-up](title="msg passed up"; source=count(IRM_PassUp);
 record=last);
@statistic[irm-down](title="msg passed down"; source=count(IRM_PassDown);
 record=last);
```

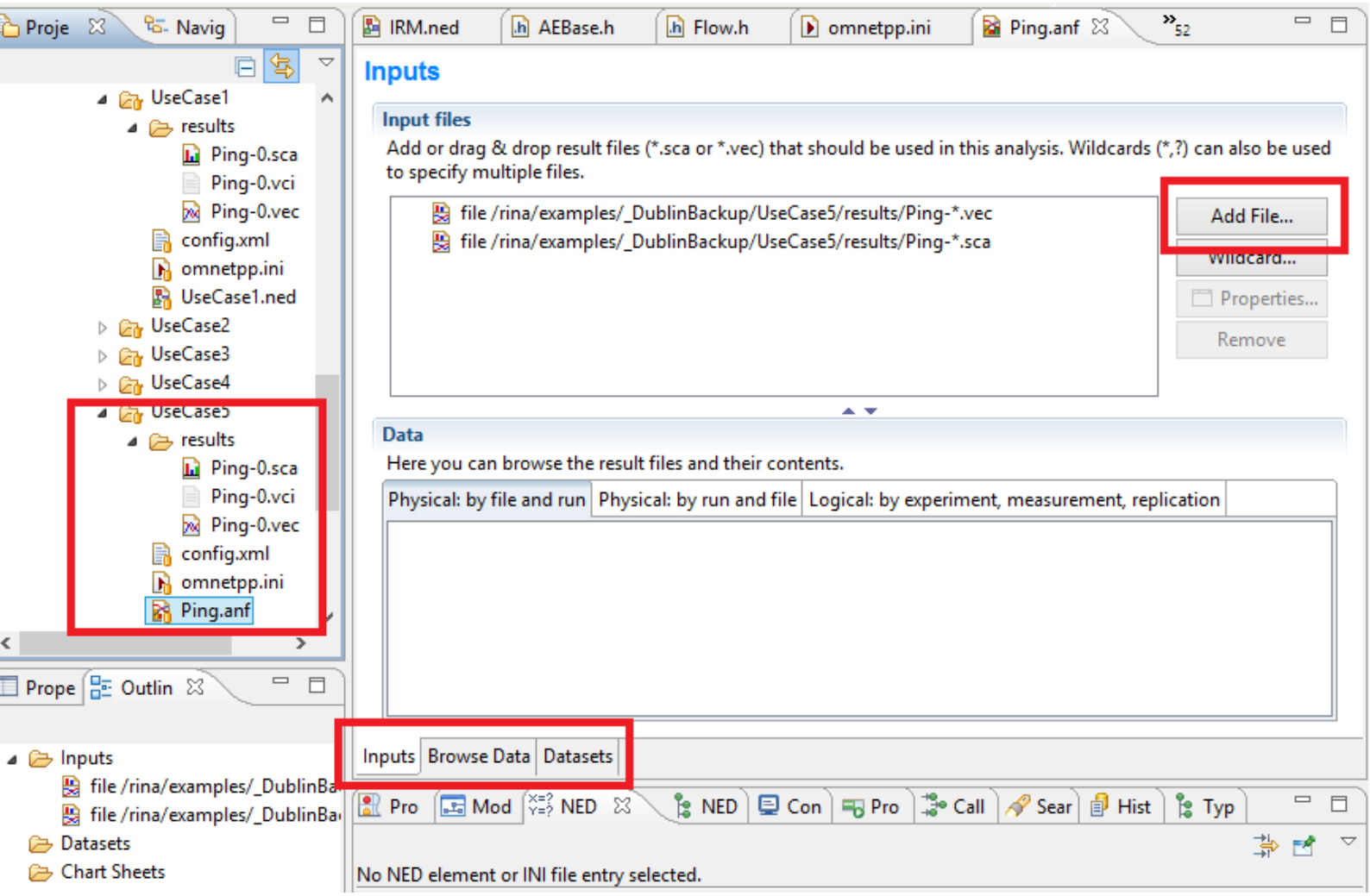

Figure 34. Results analysis

4.5.2. Tracefiles

RINASim's implementation of RMT can generate a trace file that contains a list of actions undertaken on every PDU during the simulation.

4.5.2.1. Usage

Trace file generation is enabled by the `pduTracing` parameter of the `RelayAndMux` compound module (e.g. `**.relayAndMux.pduTracing = true`).

The resulting file with a *.tr* extension is stored in the /results/ directory of chosen simulation.

4.5.2.2. Description

The lines are written in chronological order, and their format resembles that of ns-2 trace files:

```
event time node ipcp pduType pduSize flags flow DIF srcAddr dstAddr seq id
```

field	format
event	r (receive) / s (send) / + (enqueue) / - (dequeue) / d (drop)
time	event timestamp in seconds
node	node name
ipcp	IPC process name
pduType	PDU type
pduSize	PDU size in bits
flags	PDU flags
flow	flow-id (srcCEP + dstCEP + qosID)
DIF	DIF name
srcAddr	source address
dstAddr	destination address
seq	PDU sequence number
id	packet ID (unambiguous in scope of whole simulation)

4.5.2.3. Example output

```
+ 102.000346959997 interiorRouter ipcProcess0 DataTransferPDU 168 00000000
 27469590451 Layer01 3 1 5 1831
- 102.000346959997 interiorRouter ipcProcess0 DataTransferPDU 168 00000000
 27469590451 Layer01 3 1 5 1831
s 102.000346959997 interiorRouter ipcProcess0 DataTransferPDU 168 00000000
 27469590451 Layer01 3 1 5 1831
r 102.000348719997 interiorRouter ipcProcess1 DataTransferPDU 176 00000000
 216369211 Layer02 2 4 6 1804
+ 102.000348719997 interiorRouter ipcProcess1 DataTransferPDU 176 00000000
 216369211 Layer02 2 4 6 1804
- 102.000348719997 interiorRouter ipcProcess1 DataTransferPDU 176 00000000
 216369211 Layer02 2 4 6 1804
```

5. Components

This subsection provides a general overview of RINASim 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 easily orient in RINASim too. Though this structure does not always stand for the most natural representation of 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. During the design of simulation models, we were able to determine several places where specifications should be refined to provide complete and unambiguous information. Following lines reflect RINASim design relevant to the up-to-date version of RINA specifications and underlying mechanism and policies.

It is assumed that for experimenting with RINA concepts these components will be extended to the required policies depending on the character and goals of the target experiments. As mentioned in previous chapters, these components also compose predefined RINA nodes used for experimental simulation models to demonstrate properties of different RINA applications. Thus, the information provided in this chapter may be interesting to anyone who participates on RINA design and wants to perform experiments with different mechanisms and policies.

5.1. Used Template

Each atomic RINA component is described using the following set of information:

1. Visual representation of component structure
2. Narrative description of the functionality provided by the component
3. List of the component’s submodules
4. Relevant source files containing code of the component’s implementation
5. NED design structure (e.g., used dynamic and static gates, registered signals, configurable parameters and properties)
6. Available policies (a list of available user-definable policies)
7. C++ implementation notes (e.g., interface, base class, children classes, notable methods and attributes)
8. Overview of current limitations and future development plans

5.2. Nodes

RINASim offers a variety of high-level models simulating the behavior of independent computing system. These models can be employed to set quickly 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, contains 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 number of APs and AEs. The figure below illustrates some of host nodes internal structure. The most of depicted hosts contain two IPCPs, which models usual end-system with a single NIC. The host may provide only single IPCPs, which would allow IPC with only one directly connected neighbor. Alternatively, host may contain more than two IPCPs; (0)-rank IPCPs represent multiple NICs, and (1+)-rank IPCPs represent different DIFs host memberships;

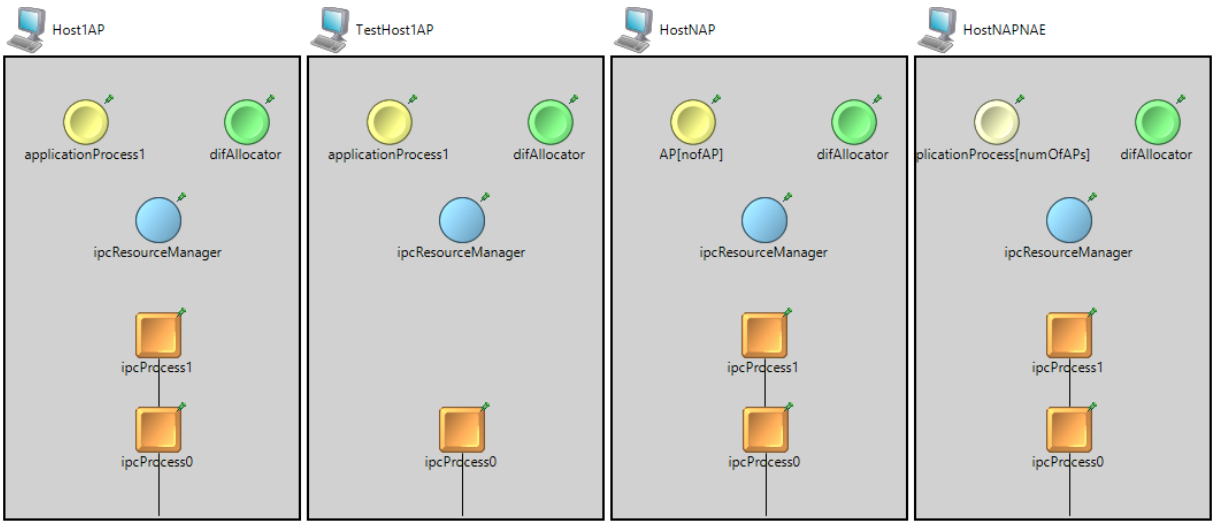

Figure 35. Host nodes structure examples

- 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 RINA specification, there are either interior or border routers depending on DIF stack depth (influenced partially also by a number of interfaces). The figure below illustrates two interior routers and one border router simulation models.

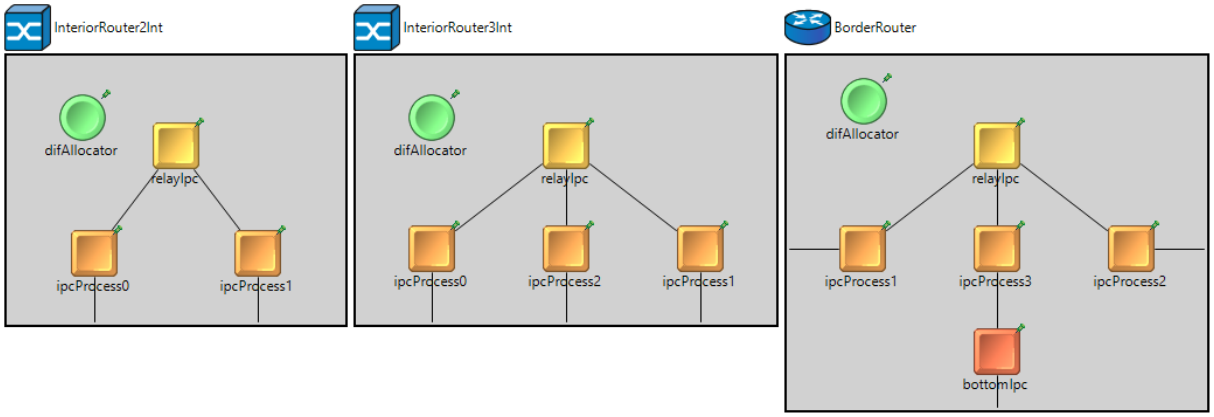

Figure 36. Router nodes structure examples

Of course, there are many more possible combinations of host and router configurations than the ones currently defined in RINASim. However, the aim of providing predefined node models is not to cover all of the possible combinations but rather to offer the most used ones enabling to set quickly up simulation scenarios. Defining new node or router with suitable structure is not a complicated task. Nevertheless, the present collection of available models seems to be enough.

5.3. DAF Modules

DAF components can be divided into three submodules: a) Application Processes (containing one or more Application Entities), which represents IPC endpoints; b) IPC Resource Manager, which interconnects APs and available IPCPs; c) DIF Allocator, which helps during APN discovery and management process. Components relationship and internal structure (described below) are depicted below.

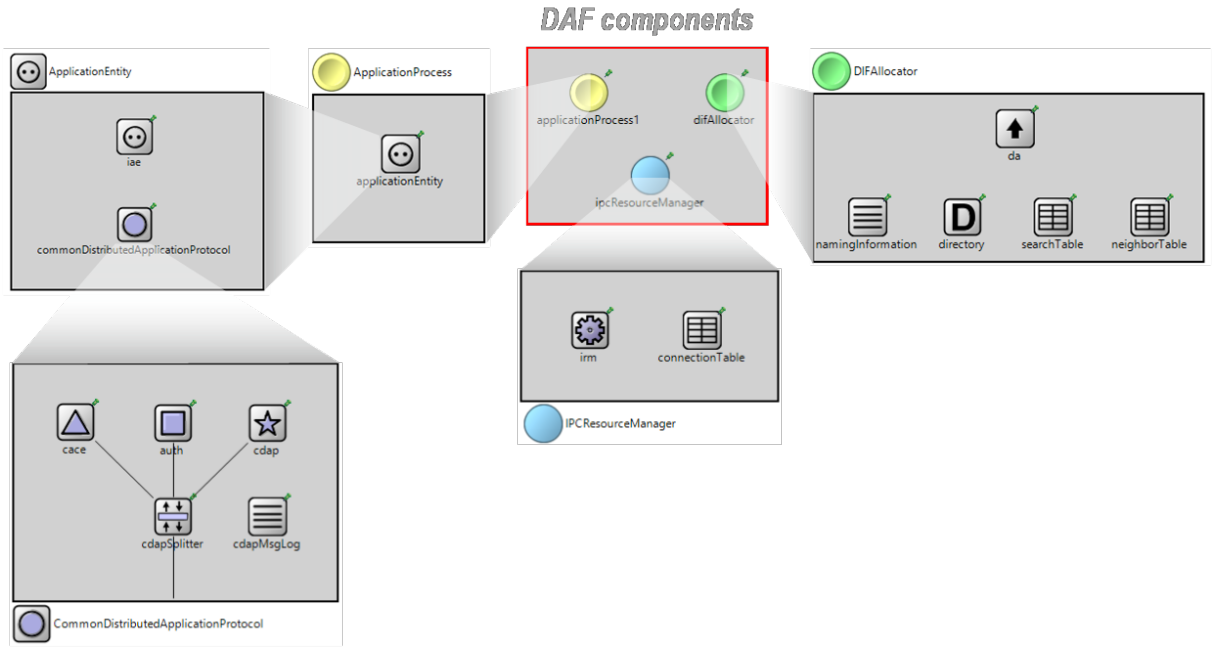

Figure 37. DAF components for RINASim

5.3.1. Application Process

The `ApplicationProcess` is a core component of DAF.Currently, this module is a placeholder for possible RINA applications.

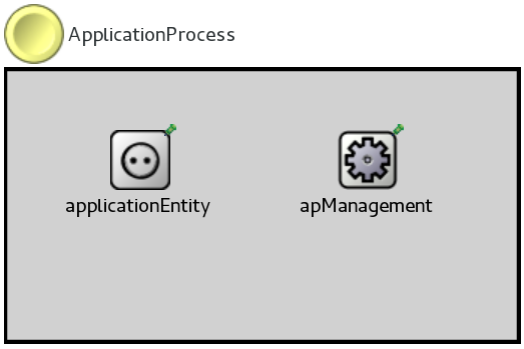

Figure 38. Application Process

5.3.1.1. Submodules

The `ApplicationProcess` modules consists of the two submodules as follows:

- `applicationEntity` – same submodule as in the case of DAF components description;;
- `apManagement` – contains Enrollment module and dynamically spawned ManagementAEs;

5.3.1.2. Source codes

Relevant sources for this component are located in */src/DAF*.

Filename(s)	Description
ApplicationProcess.ned	ApplicationProcess core simple module

5.3.1.3. NED design

- Gates utilized by this submodule are as follows:

```
applicationProcess.southIo;
applicationEntity.aeIo;
apManagement.southIo;
```

- None of ApplicationProcess has abstract data structures configurable via config.xml file.

5.3.1.4. Available policies

No policies are currently associated with this module.

5.3.1.5. C++ implementation

- This module has no signals that is receiving or emitting.

Limitations

- It is container only.

Future work

- This module should represent the core of an RINA application. As such it should be programmable instead of representing only empty container module.

5.3.2. Application Entity

The Application Entity (AE) is created for each flow representing a connection between two applications. The AE is responsible for:

- enforcing access control, i.e., to evaluate whether the requesting Application Process has access to the requested Application Process,
- monitoring and managing the associated flow during its duration.

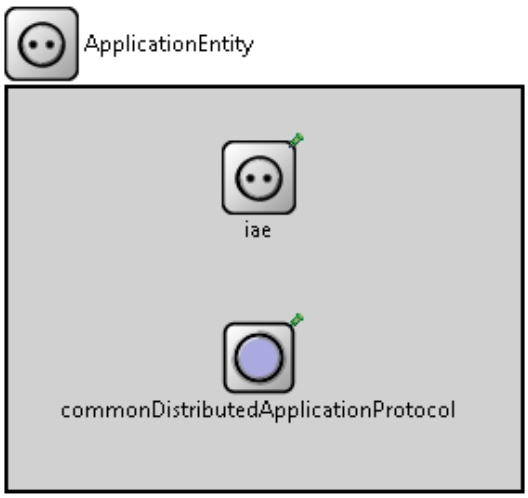

Figure 39. Application Entity

5.3.2.1. Submodules

The AE consists of two submodules:

- Interface for the AE module "iae" - AE module interface,
- Common Distributed Application Protocol module "commonDistributedApplicationProtocol". This module sends and receives CDAP messages on behalf of "iae".

5.3.2.2. Source codes

Component sources are located in /src/DAF/AE

It consists of following files:

Filename(s)	Description
ApplicationEntity.ned	Compound module holding all the AE functionality submodules
IAE.ned	OMNeT++ NED interafce definition
AEBase.h/.cc	Base class for general AE functionality intended for inheritance and extensions
AE.ned	AE simple module generally with one-flow scheduling flow (de)allocation
AE.h/.cc	Implementation of AE core functionality
AEListeners.h/cc	AE listeners
AEPing.ned	AEPing simple module
AEPing.h/.cc	AE with Ping-like application behavior

5.3.2.3. NED design

The IAE is specified before implementation starts. Default AE type is AE.ned.

```
parameters:
 string aeType = default("AE");
submodules:
 iae: <aeType> like IAE
```

5.3.2.4. C++ Implementation

Registered signals that the AE module is emitting:

```
SIG_AE_AllocateRequest
SIG_AE_DeallocateRequest
SIG_AE_DataSend
```

```
SIG_AERIBD_AllocateResponsePositive
SIG_AERIBD_AllocateResponseNegative
```

Registered signals that the AE module is receiving:

```
SIG_AE_Enrolled
SIG_CDAP_DateReceive
SIG_FAI_AllocateRequest
SIG_FAI_DeallocateRequest
SIG_FAI_DeallocateResponse
SIG_FAI_AllocateResponsePositive
SIG_FAI_AllocateResponseNegative
```

5.3.2.5. Future work

1. Revisiting the interfaces would be necessary to adjust interfaces to recent development.
2. Create new streaming application capable of congesting the resources allocated for the flow within the DIF.

5.3.3. DAFEnrollment

The `DAFEnrollment` module controls initial communication between two IPCP's, Flow allocation and dynamic Application Entity creation and finalization.

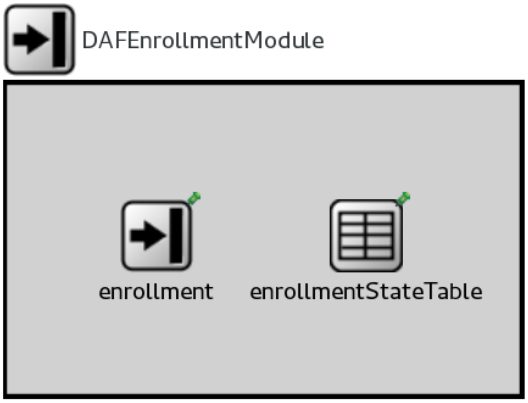

Figure 40. DAF Enrollment

5.3.3.1. Submodules

The `DAFEnrollment` modules consists of two auxiliary submodules that maintain the state information:

- `enrollment` – this module implements the core functionality;
- `enrollmentStateTable` – this module maintains status of active flows;

5.3.3.2. Source codes

Relevant sources for this component are located in */src/DAF/Enrollment*.

Filename(s)	Description
EnrollmentModule.ned	DAFEnrollment compound module that is part of every node
Enrollment.ned	DAFEnrollment core simple module
DAFEnrollment.h/.cc	Implementation of DAFEnrollment core functionality
DAFEnrollmentBase.h/.cc	Base class for general DAFEnrollment functionality intended for inheritance and extensions
EnrollmentStateTable.ned	State table simple module
DAFEnrollmentStateTable.h/.cc	Implementation of state table functionality
DAFEnrollmentStateTableEntry.h/.cc	Single record for state table, basically destination APN as key and source APN, CACE Connection status, Enrollment status
DAFEnrollmentObj.h/.cc	Implementation of DAFEnrollment object maintaining information exchanged during enrollment phase
DAFOperationObj.h/.cc	Implementation of DAFEnrollment object maintaining information exchanged after enrollment phase
DAFEnrollmentListeners.h/.cc	Enrollment listeners
DAFEnrollmentNotifierBase.h/.cc	Base class for general DAFEnrollment Notifier functionality intended for inheritance and extensions
DAFEnrollmentNotifier.h/.cc	Implementation of DAFEnrollment Notifier core functionality
DAFEnrollmentNotifierListeners.h/.cc	DAFEnrollment Notifier listeners

5.3.3.3. NED design

- DAFEnrollment and its submodules do not have any gates. The module communicates using signals only.

- None of DAFEnrollment has abstract data structures configurable via config.xml file.

5.3.3.4. Available policies

No policies are currently associated with this module.

5.3.3.5. C++ implementation

- Registered signals that the DAFEnrollment and its submodules are emitting:

```
SIG_ENROLLMENT_CACEDataSend
SIG_ENROLLMENT_DataSend
SIG_ENROLLMENT_StartEnrollmentRequest
SIG_ENROLLMENT_StartEnrollmentResponse
SIG_ENROLLMENT_StopEnrollmentRequest
SIG_ENROLLMENT_StopEnrollmentResponse
SIG_ENROLLMENT_StartOperationRequest
SIG_ENROLLMENT_StartOperationResponse
SIG_ENROLLMENT_Finished
SIG_AEMGMT_ConnectionResponsePositive
SIG_AERIBD_AllocateResponseNegative
SIG_AERIBD_AllocateResponsePositive
```

- Registered signals that the DAFEnrollment and its submodules are receiving:

```
SIG_FAI_AllocateResponsePositive
SIG_FAI_AllocateRequest
SIG_AE_Enrolled
SIG_RIBD_StartEnrollmentRequest
SIG_RIBD_StartEnrollmentResponse
SIG_RIBD_StopEnrollmentRequest
SIG_RIBD_StopEnrollmentResponse
SIG_RIBD_StartOperationRequest
SIG_RIBD_StartOperationResponse
SIG_RIBD_ConnectionResponsePositive
SIG_RIBD_ConnectionResponseNegative
SIG_RIBD_ConnectionRequest
```

5.3.3.6. Limitations

- This module does not support deallocation.
- The module cannot be configured.

5.3.3.7. Future work

1. Deallocation of the module should be supported.
2. An implementation that allows to send application-specific data in DAF enrollment phase should be provided.

5.3.4. DIF Allocator

The `difAllocator` module handles locating a destination application based on its name. DIF Allocator is a component of the DAP's IPC Management that takes 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 simulation network graph.

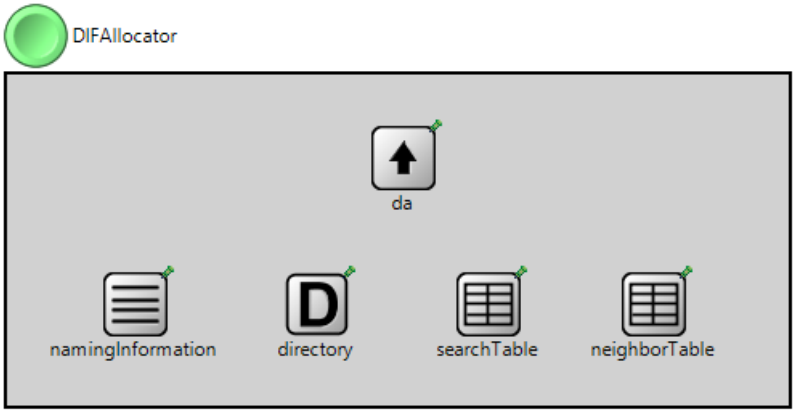

Figure 41. DIF Allocator

5.3.4.1. Submodules

The `difAllocator` modules consists of five auxiliary submodules that maintain state information:

- `da` – core functionality;
- `namingInformation` – mapping between APN synonyms;
- `directory` – mapping between APN and DIF-names;
- `searchTable` – mapping between APN and peer DA instance where to continue search;
- `neighborTable` – mapping between peer DA and neighboring DA instances.

5.3.4.2. Source codes

Relevant sources for this component are located in */src/DAF/DA*.

Filename(s)	Description
DIFAllocator.ned	DIF Allocator compound module that is part of every node
DA.ned	DA core simple module
DA.h/.cc	Implementation of DA core functionality
NamingInformation.ned	Synonyms naming table simple module
NamingInformation.h/.cc	Implementation of Synonyms naming table functionality
NamingInformationEntry.h/.cc	Single record for naming table, basically APN as key and list of assigned synonyms (other APNs)
Directory.ned	Directory mapping simple module
Directory.h/.cc	Implementation of Directory mapping functionality
DirectoryEntry.h/.cc	Single directory record, which contains APN as primary key and list of Addresses
SearchTable.ned"	Searching table simple module
SearchTable.h/.cc	Implementation of Searching table functionality
SearchTableEntry.h/.cc	Implementation of Auth core functionality
NeigborTable.ned"	Neighbor table simple module
NeigborTable.h/.cc	Implementation of Neighbor table functionality
NeigborTableEntry.h/.cc	Implementation of Auth core functionality

5.3.4.3. NED design

- DIF Allocator and its submodules do not have any gates.
- DIF Allocator and its submodules abstract data structures are configurable via *config.xml* file.

5.3.4.4. Available policies

No policies are currently associated with this module.

5.3.4.5. C++ implementation

- DIF Allocator does not receive/emit any signals. Usage of DIF Allocator components is done via direct function calls.

5.3.4.6. Limitations

- SearchTable does not have any impact on current RINASim functionality.

5.3.4.7. Future work

1. Define interface for DIF allocator;
2. The content of NeighborTable should not be used for FA delivery. Use dynamic Routing information instead.

5.3.5. IPC Resource Manager

The `ipcResourceManager` module currently queries DA module to find suitable IPCP and relays communication between AE and IPCP. The `ipcResourceManager` consists of two submodules:

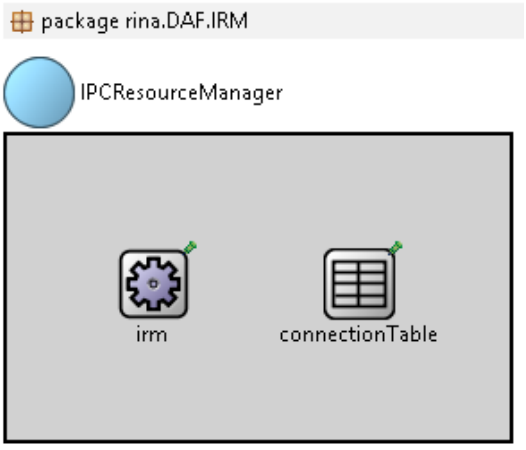

Figure 42. IPC Resource Manager

5.3.5.1. Submodules

The IPC Resource Manager consists of two submodules:

- `irm` - This module acts as a broker between APs and IPCs and handles AP flow (de)allocation calls
- `connectionTable` - This module maintains the necessary state for IRM proper functionality (the state of the N-1 flows).

5.3.5.2. Source codes

Component sources are located in */src/DAF/IRM*. It consists of following files:

Filename(s)	Description
IPCResourceManager.ned	IPC Resource Manager compound module that is part of Host nodes
IRM.ned	IRM simple module
IRM.h/.cc	Implementation of IRM core functionality
IRMListeners.h/cc	Listeners that catches signals, which IRM should process
ConnectionTable.ned	Connection Table simple module
ConnectionTable.h/.cc	Connection Table implementation as a table storing state of AP communication
ConnectionTableEntry.h/.cc	Single Connection Table entry with all its properties

5.3.5.3. NED design

- IRM and its submodules utilizes following gates:

```
IPCResourceManager.northIo
IRM.aeIo
IRM.southIo_
IPCResourceManager.southIo
```

- None of IRM submodules has abstract data structures configurable via *config.xml* file.

5.3.5.4. Available policies

No policies are currently associated with this module.

5.3.5.5. C++ Implementation

- Registered signals that IRM module is emitting:

```
IRM-AllocateRequest
IRM-DeallocateRequest
```

- IRM handles direct API calls from AP, mainly the ones that are related to the flow (de)allocation data-path.

5.3.5.6. Future work

1. Define interfaces for both IRM and Connection Table;
2. Change "IRM.aeIo" gate name to something more meaningful.

5.3.6. Common Distributed Application Protocol

The `commonDistributedApplicationProtocol` submodule provides a simple object-based protocol for distributed applications.

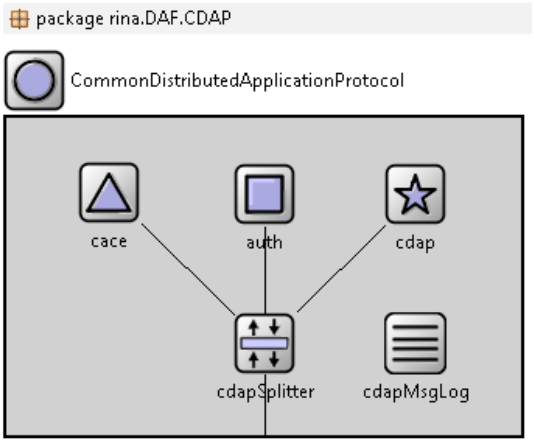

Figure 43. CDAP module

5.3.6.1. Submodules

Currently, it is the part of RIBd and AE. CDAP is modeled as a compound module consisting of five main submodules:

- `cace` – Common Application Connection Establishment protocol instance processing *M_CONNECT* and *M_RELEASE* requests and responses;
- `auth` – providing authentication services during connection initialization); cdap (providing usual CDAP message exchange;
- `cdapSplitter` – delivering messages to appropriate upper submodules;
- `cdapMsgLog` – logger for an accounting of processed messages.

5.3.6.2. Source code

Relevant sources for this component are located in */src/DAF/CDAP*.

Filename(s)	Description
CommonDistributedApplicationProtocol.ned	CDAP compound module that is part of ApplicationEntity and RIBDaemon modules
CACE.ned	CACE simple module
CACE.h/.cc	Implementation of CACE core functionality
CACEListeners.h/.cc	Listeners that catch signals during enrollment procedure
Auth.ned	Auth simple module
Auth.h/.cc	Implementation of Auth core functionality
AuthListeners.h/.cc	Listeners that catch signals during enrollment procedure
CDAP.ned	CDAP simple module
CDAP.h/cc	Implementation of CDAP core functionality
CDAPListeners.h/cc	Listeners that catch signals, which CDAP later processes
CDAPSplitter.ned	CDAP splitter module
CDAPSplitter.h/cc	Implementation of a CDAP splitter that forwards them to the appropriate CDAP module according to the CDAP message type.
CDAPMsgLog.ned	CDAP simple module
CDAPMsgLog.h/cc	Implementation of CDAP message logger functionality which records incoming/ outgoing messages that pass through "cdapSplitter".
CDAPMsgLogEntry.h/cc	Single CDAP message logger entry with all of its properties
CDAPMessage.msg	OMNeT++ CDAP message definition file
CDAPMessage_m.h/.cc	C++ implementation of CDAP message classes

5.3.6.3. NED design

- CDAP module contains seed invokationId parameter.
- Data-path of interconnected gates for messages:

```
cdapSplitter.caceIo
cdapSplitter.authIo
cdapSplitter.cdapIo
cdapSplitter.southIo
caceIo.splitterIo
authIo.splitterIo
cdapIo.splitterIo
```

- None of CDAP submodules has abstract data structures configurable via *config.xml* file.

5.3.6.4. Available policies

No policies are currently associated with CDAP and its submodules.

5.3.6.5. C++ implementation

- Registered signals that the CDAP and its submodules are emitting:

```
SIG_CDAP_DateReceive
SIG_CACE_DataReceive
```

- Registered signals that the CDAP and its submodules are processing:

```
SIG_AE_DataSend
SIG_RIBD_DataSend
SIG_RIBD_CACESend
```

5.3.6.6. Limitations

1. Auth module is currently still placeholder.

5.3.6.7. Future work

1. Together with AE define CDAP call API.

5.4. DIF Modules

All currently implemented DIF components are enclosed to the IPCProcess container module (instantiation of IPCP). The IPCProcess contains following submodules, and overall structure is shown in below:

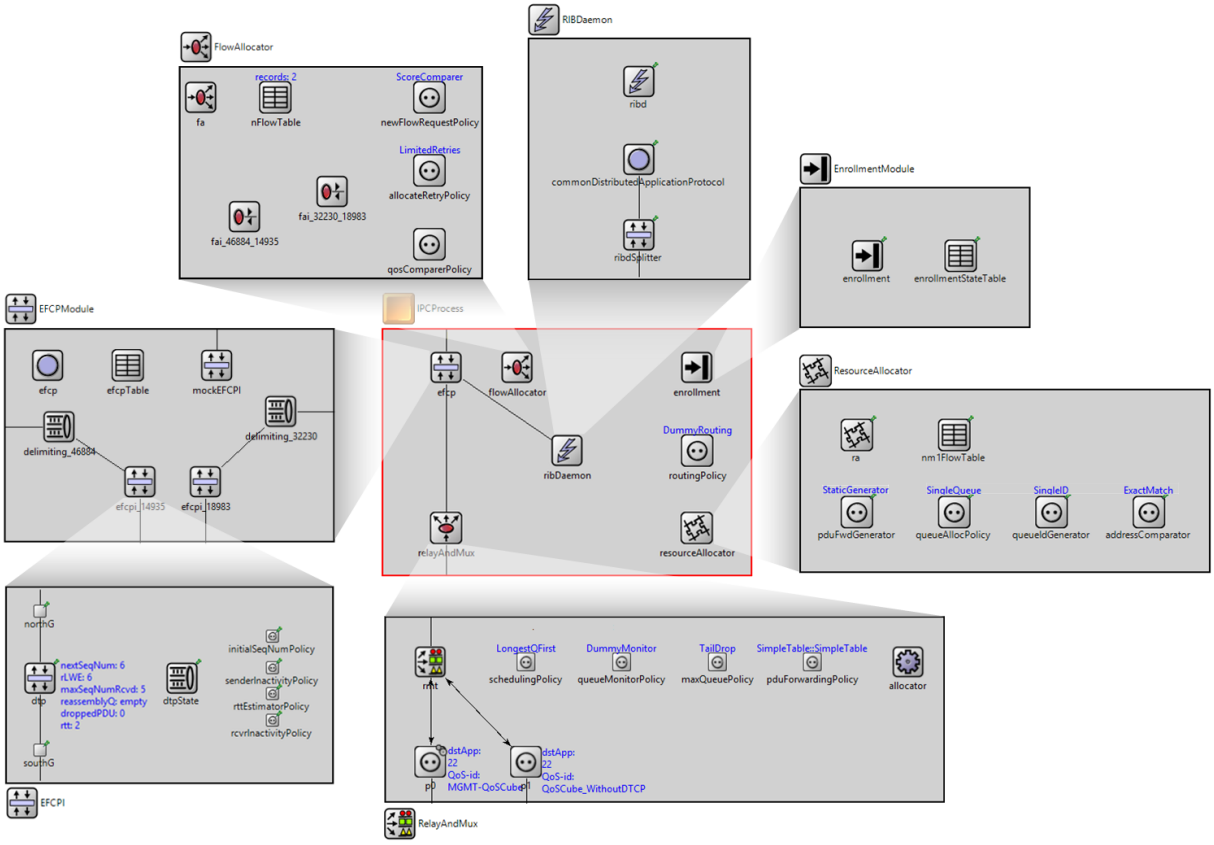

Figure 44. IPCP's DIF components for RINASim

5.4.1. Delimiting

The delimiting module handles SDUs in the form of SDUData from N+1 DIF and produces UserDataField for EFCP module. In the opposite direction, it accepts UserDataFieldD and produces SDUData to N+1 DIF. Encapsulation process is done according to Delimiting process and uses these classes: SDUData $\rightarrow$ Data $\rightarrow$ PDUdata $\rightarrow$ UserDataField

It is capable of fragmentation and concatenation of incoming SDUs. Fragmentation is based on maxPDUsize. Concatenation takes incoming SDUs and puts them in single PDUData until maxPDUSize is met or until Delimiting Timer expires. If SDU with the size smaller than maxPDUSize is received or fragment is generated, the DelimitingTimer is scheduled. DelimitingTimer specifies the maximum time the SDU can be retained in an attempt to concatenate it with subsequent SDU. SDUs that size is bigger than maxPDUSize*0.8 are not held, and it is processed immediately.

PDUData class has overridden methods for encapsulate()/decapsulate to take/ return Data class. Moreover, it is possible to encapsulate several Data packets into one PDUData.

The SDU marked as the first fragment contain the whole SDU, and the rest of the fragments are empty. All non-first fragments are deleted upon de-fragmentation, but the first_fragment SDU is passed to N+1 DIF only if all fragments are present.

5.4.1.1. Submodules

The delimiting module does not contain any submodule.

5.4.1.2. Source codes

Relevant sources for this component are located in /src/DIF/Delimiting/.

Filename(s)	**Description**
Data.cc/h	Enhanced implementation of generated Data packet class.
Data.msg	Message definition for representing SDUs and SDU fragments.
Delimiting.cc/h	Implementation of delimiting functions.
Delimiting.ned	Delimiting module
DelimitingTimers.msg	Timers related to delimiting
PDUData.cc/h	Enhanced msg class for encapsulating multiple messages.
PDUData.msg	Message class for PDUData
UserDataField.h/.cc	Implementation of User Data Field message.
UserDataField.msg	Message class for UserDataField

5.4.1.3. NED design

- Delimiting unitilizes following gates: northIo; //towards FAI southIo[0]; //towards DTP
- Delimiting module does not have abstract data structures configurable via config.xml file.

5.4.1.4. Available policies

No policies are currently associated with this module.

5.4.1.5. C++ implementation

- Delimiting does not receive/emit any signals.

5.4.1.6. Limitations

- Delimiting expects that the received UserDataField are in order and complete.
- Delimiting does not control maxSDUsize on incoming data from N+1 DIF.

5.4.1.7. Future work

1. To enable replaceable policies for incoming SDUs from N+1 DIF and User Data Fields from EFCP.

5.4.2. Enrollment

The `Enrollment` module controls initial communication (enrollment phase) between two IPCP's. It contains functionality for IPCP address assignment.

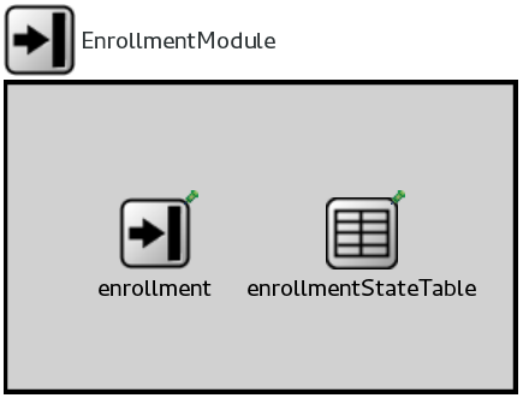

Figure 45. Enrollment

5.4.2.1. Submodules

The `Enrollment` modules consists of two submodules that provide the functionality and maintains the state information during enrollment:

- `enrollment` – implements the core functionality;
- `enrollmentStateTable` – maintains states of active flows;

5.4.2.2. Source codes

Relevant sources for this component are located in */src/DIF/Enrollment*.

Filename(s)	**Description**
EnrollmentModule.ned	Enrollment compound module that is part of every node
Enrollment.ned	Enrollment core simple module
Enrollment.h/.cc	Implementation of Enrollment core functionality

Filename(s)	Description
EnrollmentBase.h/.cc	Base class for general Enrollment functionality intended for inheritance and extensions
EnrollmentStateTable.ned	State table simple module
EnrollmentStateTable.h/.cc	Implementation of state table functionality
EnrollmentStateTableEntry.h/.cc	Single record for state table, basically destination APN as key and source APN, CACE Connection status, Enrollment status
EnrollmentObj.h/.cc	Implementation of Enrollment object maintaining information exchanged during enrollment phase
OperationObj.h/.cc	Implementation of Enrollment object maintaining information exchanged after enrollment phase
EnrollmentListeners.h/.cc	Enrollment listeners
EnrollmentNotifierBase.h/.cc	Base class for general Enrollment Notifier functionality intended for inheritance and extensions
EnrollmentNotifier.h/.cc	Implementation of Enrollment Notifier core functionality
EnrollmentNotifierListeners.h/.cc	Enrollment Notifier listeners

5.4.2.3. NED design

- Enrollment and its submodules do not have any gates. They communicate using signals.
- Enrollment and its submodules abstract data structures are configurable via config.xml file.

5.4.2.4. Available policies

No policies are currently associated with this module.

5.4.2.5. C++ implementation

- Registered signals that the Enrollment module and its submodules are emitting consist of following:

```
SIG_ENROLLMENT_CACEDataSend
SIG_ENROLLMENT_DataSend
SIG_ENROLLMENT_StartEnrollmentRequest
SIG_ENROLLMENT_StartEnrollmentResponse
SIG_ENROLLMENT_StopEnrollmentRequest
SIG_ENROLLMENT_StopEnrollmentResponse
SIG_ENROLLMENT_StartOperationRequest
SIG_ENROLLMENT_StartOperationResponse
SIG_ENROLLMENT_Finished
```

- Registered signals that the Enrollment and its submodules are receiving consists of following:

```
SIG_FA_MgmtFlowAllocated
SIG_RIBD_StartEnrollmentRequest
SIG_RIBD_StartEnrollmentResponse
SIG_RIBD_StopEnrollmentRequest
SIG_RIBD_StopEnrollmentResponse
SIG_RIBD_StartOperationRequest
SIG_RIBD_StartOperationResponse
SIG_RIBD_ConnectionResponsePositive
SIG_RIBD_ConnectionResponseNegative
SIG_RIBD_ConnectionRequest
```

5.4.2.6. Limitations

- This module does not implemented deallocation functionality.

5.4.2.7. Future work

1. Module deallocation implementation should be provided.
2. Enrollment module does not send user specified data. For some scenarios, it would be useful to have ability sending custom data from Enrollment module.

5.4.3. Error and Flow Control Compound module

The `efcp` compound module handles data transfer and associated state vectors. It takes SDUs from N+1 or CDAP message from RIB Daemon and creates complete PDUs.

The Error and Flow Control Protocol (EFCP) is modeled as one compound module. This module dynamically generates EFCP Instances. Dynamic modules consist of one Delimiting module

and (possibly) multiple EFCPI modules per one flow. The `EFCPI` module itself is a compound module and contains static modules `DTP` and `DTPState`, and if the flow (QoS requirements) requires *control*, then there are `DTCP` and `DTCPState` modules.

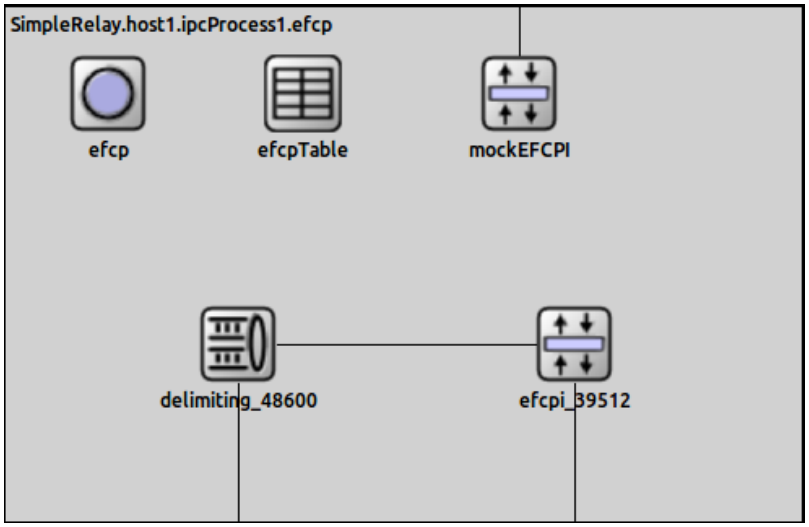

Figure 46. EFCP module with dynamically created Delimiting and EFCP instance modules

5.4.3.1. Submodules

The `efcp` modules consists of three static submodules:

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

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

- `delimiting_<portId>` – handles fragmentation/concatenation
- `efcpi_<cepId>` – handles data transfer + control fuctions

5.4.3.2. Source codes

Relevant sources for this component are located in */src/DIF/EFCP*.

Filename(s)	Description
EFCPTable/	Implementation and .ned module definition for EFCPTable
DTP/	All files related to DTP[2]

[2] D26-RINASim-DTP

Filename(s)	Description
DTCP/	All files related to DTCP[3]
EFCP_defs.h	Definitions and constants related only to EFCP.
EFCP.cc/h	Implementaion of static EFCP modules governing creation and deletion of dynamic modules.
EFCPInstance.cc/h	Couples together DTP and DTCP.
EFCPListeners.cc/h	Implementation of EFCP's singnal listeners
EFCPPolicySet.cc/h	Class defining set of EFCP policies for QoSCube.
ManagementPDU.msg	Message definition for Management PDUs.
MockEFCPI.cc/h	Simplified EFCPI module for Management PDUs.
MockEFCPI.ned	Simplified EFCPI simple module.

5.4.3.3. NED design

- EFCP compound module utilizes these static gates:

```
ribd
mockToRMT
```

Besides static gates, there are two dynamically created gates per every active flow.

```
northIo_<portId>
soutIo_<cepId>
```

Full data-path of interconnected gates for messages going through EFCP Compound module then looks as follows:

```
northIo - towards ipc northIo
delimiting_<portId>.northIo_<portId>
delimiting_<portId>.southIo_<portId>
efcpi_<cep>.northIo
```

[3] D26-RINASIM-DTCP

```
efcpi_<cep>.southIo
southIo - towards RMT
```

- EFCP Compound module is not configurable via *config.xml* file.

5.4.3.4. Available policies

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

5.4.3.5. C++ implementation

- EFCP Compound module does not receive/emit any signals. Usage of DIF Allocator components is done via direct function calls.

5.4.3.6. Future work

1. To provide a better visualisation of dynamically created modules.

5.4.4. EFCP Instance

An EFCP Instance locally manages established flow. The `efcpi_<cepId>` module contains `DTP` and `DTPState` submodules. If QoS requires more control on the flow, e.g., reliable data delivery, this module also contains `DTCP` and `DTCPState` submodules. Also, any necessary policy submodules associated with the flow are also part of this compound module.

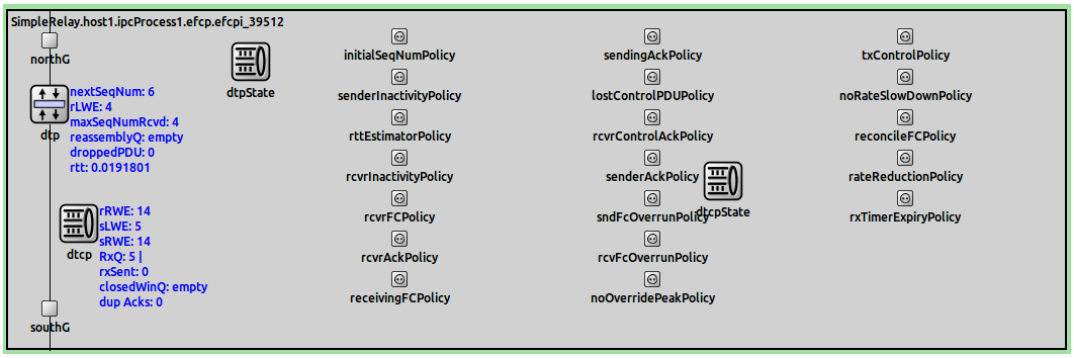

Figure 47. EFCP Instance

5.4.4.1. Submodules

The EFCP Instance consists of the following submodules:

- `dtp` - This module provides implementation of Data Transfer Protocol.

- `dtpState` - This module holds DTP related variables.
- `dtcp` - This module provides implementation of Data Transfer Control Protocol.
- `dtcpState` - This module maintains DTCP related variables.
- `<policyName>Policy` - Module that represents a specific policy. Currently there is no such module.
- `northG`, `southG` - Pass-through modules that serves for better link visualisation.

5.4.4.2. Source codes

The EFCP Instance module is a compound module and hence it does not have any implementation.

Filename(s)	Description
EFCPI.ned	EFCPI module definition.

5.4.4.3. NED design

- Data-path of interconnected gates for messages going through EFCPI module:

```
efcpi_<cepId>.northIo - towards delimiting
efcpi_<cepId>.northG.northIo
efcpi_<cepId>.northG.southIo
efcpi_<cepId>.dtp.northIo
efcpi_<cepId>.dtp.southIo
efcpi_<cepId>.southG.northIo
efcpi_<cepId>.southG.southIo
efcpi_<cepId>.southIo - towards EFCP Compound Module southIo
```

- The submodules of EFCPI contains abstract data structures that are configurable via *config.xml* file.

5.4.4.4. Available policies

No policies are currently associated with this module.

5.4.4.5. C++ Implementation

- EFCPI does not receive/emit any signals.

5.4.5. DTP

The `dtp` module accepts user data content from the Delimiting module, generates PDUs, and pass them to RMT. If necessary, it asks DTCP to perform Retransmission and Flow Control. A-Time value can parametrize the current DTP implementation.

Figure 48. Data Transfer Protocol module

5.4.5.1. Submodules

The DTP module does not have any submodules.

5.4.5.2. Source codes

Component sources are located in */src/DIF/EFCP/DTP/*. They consist of following files:

Filename(s)	**Description**
DTP.ned	DTP module
DTP.cc/h	DTP functionality
DTPTimers.msg	Timers related to DTP module
DataTransferPDU.msg	Message definition for Data Transfer PDU
DataTransferPDU.cc/h	Extended implementation of generated `DataTransferPDU` class
DumbGate.ned	Dumb gate module
DumbGate.cc/h	Implementation of a pass-through gate module.

5.4.5.3. NED design

- Data-path of interconnected gates for messages going through EFCPI:

```
efcpi_<cepId>.northIo
northIo - towards EFCPI's northIo
southIo - towards EFCPI's southIo
```

```
efcpi_<cepId>.southIo
```

- DTP policies are configurable via *config.xml* file by specifying EFCP Policy Set in QoS Cube. None of IRM submodules has abstract data structures configurable via *config.xml* file.

5.4.5.4. Available policies

The DTP module can be extended with the following policies:

- `InitialSeqNumPolicy` - see subchapter 6.2.1[4]
- `RcvrInactivityPolicy` - see subchapter 6.2.2[5]
- `SenderInactivityPolicy` - see subchapter 6.2.3[6]
- `RTTEstimatorPolicy` - see subchapter 6.2.4[7]

5.4.5.5. C++ Implementation

- Registered signals that IRM module is emitting:

```
EFCP-StopSending
EFCP-StartSending](type=Flow?);
DTP_RTT
DTP_CLOSED_WIN_Q
```

- DTP handles direct API calls from DTPState and DTCP modules, mainly the ones that are related to the actual sending of PDU.

5.4.5.6. Limitations

- The current implementation of DTP does not support partial delivery as this policy would require full implementation of PDU serialization.

5.4.5.7. Future work

1. A proper place for RTTEstimator policy needs to be determined. DTP accepts this policy, but the necessary information is in DTCP.

[4] D26-RINASim-Policies-EFCP-InitialSequenceNumber
[5] D26-RINASim-Policies-EFCP-RcvrTimerInactivity
[6] D26-RINASim-Policies-EFCP-SenderTimerInactivity
[7] D26-RINASim-Policies-EFCP-RTTEstimator

5.4.6. DTP State

The `dtpState` (DTP-SV) holds properties related to the actual data transfer. In RINASim, the `dtpState` module stores all necessary variables such as `rcvLeftWindowEdge` and `nextSeqNumToSend` plus queues for generated and postable PDUs and reassembly queue.

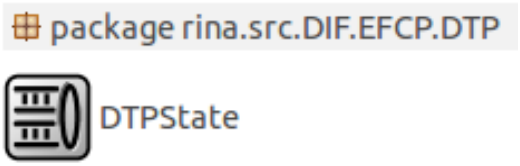

Figure 49. DTP State module

5.4.6.1. Submodules

The DTP module does not have any submodules.

5.4.6.2. Source codes

Component sources are located in */src/DIF/EFCP/DTP/*. It consists of following files:

Filename(s)	Description
DTPstate.ned	DTP State simple module
DTPState.cc/h	DTP State implementation

5.4.6.3. NED design

- DTP State module does not have any gates.
- DTP State module is not configurable via *config.xml* file.

5.4.6.4. Available policies

The DTP State module does not have any policy.

5.4.6.5. C++ Implementation

- DTP State module does not emit any signals.
- DTP handles direct API calls from DTP, DTCP, and related policies.

5.4.7. DTCP

The `dtcp` handles retransmission and flow control related tasks. From the perspective of RINASim, DTCP is a module that runs policies to update the `dtcpState`. Policies implement different reactions to a situation when error recovery and flow control is expected.

The current implementation supports retransmission, window-based flow control, allowed gap, A-Time. Besides capabilities defined in specifications it supports following features:

- Rendezvous Mechanism - recovery mechanism for lost control information about opened window
- ECN SlowDown - In cooperation with RA and RMT it may receive backward ECN information from the relay node.

Figure 50. Data Transfer Control Protocol module

5.4.7.1. Submodules

The DTCP module does not have any submodules.

5.4.7.2. Source codes

Component sources are located in */src/DIF/EFCP/DTCP/*. It consists of following files:

Filename(s)	Description
DTCP.ned	DTCP module
DTCP.cc/h	DTCP functionality
DTCPTimers.msg	Timers related to DTCP module
ControlPDU.msg	Definition of Control PDUs used in Flow Control, Retransmission, and Rendezvous mechanism

5.4.7.3. NED design

- DTCP module does not have any gates.

- DTCP module is not configurable via *config.xml* file.

5.4.7.4. Available policies

The DTCP module is associated with following policies:

- `ECNPolicy` - see subchapter 6.2.5[8]
- `ECNSlowDownPolicy` - subchapter 6.2.6[9]
- `LostControlPDUPolicy` - see subchapter 6.2.7[10]
- `NoOverridePeakPolicy` - see subchapter 6.2.8[11]
- `NoRateSlowDownPolicy` - see subchapter 6.2.9[12]
- `RateReductionPolicy` - see subchapter 6.2.10[13]
- `RcvFCOverrunPolicy` - see subchapter 6.2.11[14]
- `RcvrAckPolicy` - see subchapter 6.2.12[15]
- `RcvrControlAckPolicy` - see subchapter 6.2.13[16]
- `RcvrFCPolicy` - see subchapter 6.2.14[17]
- `ReceivingFCPolicy` - see subchapter 6.2.15[18]
- `ReconcileFCPolicy]` - see subchapter 6.2.16[19]
- `RxTimerExpiryPolicy]` - see subchapter 6.2.17[20]
- `SenderAckPolicy]` - see subchapter 6.2.18[21]
- `SenderAckListPolicy]` - see subchapter 6.2.19[22]
- `SendingAckPolicy]` - see subchapter 6.2.20[23]

[8] D26-RINASim-Policies-EFCP-ECN
[9] D26-RINASim-Policies-EFCP-ECNSlowDown
[10] D26-RINASim-Policies-EFCP-LostControlPDU
[11] D26-RINASim-Policies-EFCP-NoOverrideDefaultPeak
[12] D26-RINASim-Policies-EFCP-NoRate-SlowDown
[13] D26-RINASim-Policies-EFCP-RateReduction
[14] D26-RINASim-Policies-EFCP-RcvFlowControlOverrun
[15] D26-RINASim-Policies-EFCP-RcvrAck
[16] D26-RINASim-Policies-EFCP-RcvrControlAck
[17] D26-RINASim-Policies-EFCP-RcvrFlowControl
[18] D26-RINASim-Policies-EFCP-ReceivingFlowControl
[19] D26-RINASim-Policies-EFCP-ReconcileFlowConflict
[20] D26-RINASim-Policies-EFCP-RetransmissionTimerExpiry
[21] D26-RINASim-Policies-EFCP-SenderAck
[22] D26-RINASim-Policies-EFCP-SenderAckList
[23] D26-RINASim-Policies-EFCP-SendingAck

- `SndFCOverrunPolicy]` - see subchapter 6.2.21[24]
- `TxControlPolicy]` - see subchapter 6.2.22[25]

5.4.7.5. C++ Implementation

- Registered signals that DTCP module is emitting:

```
DTCP_RX_SENT
```

- DTCP handles direct API calls from DTCPState and DTP modules.

Future work

1. Finishing rate-based Flow Control

Back to table of contents[26]

5.4.8. DTCP State

The `dtcpState` (DTCP-SV) holds properties related to the *control* part of data transfer. In RINASim, the `dtcpState` module stores the Retransmission queue and the Closed window queue.

Figure 51. DTCP State module

5.4.8.1. Submodules

The DTCP module does not have any submodules.

5.4.8.2. Source codes

Component sources are located in */src/DIF/EFCP/DTCP/*. It consists of following files:

[24] D26-RINASim-Policies-EFCP-SndFlowControlOverrun
[25] D26-RINASim-Policies-EFCP-TransmissionControl
[26] D26-Table-of-Contents

Filename(s)	Description
DTCPstate.ned	DTCP State simple module
DTCPState.cc/h	DTCP State implementation

5.4.8.3. NED design

- DTCP State module does not have any gates.
- DTCP State module is not configurable via *config.xml* file.

5.4.8.4. Available policies

The DTCP State module does not have any policy.

5.4.8.5. C++ Implementation

- DTCP State module does not emit any signals.
- DTCP handles direct API calls from DTCP and related policies.

5.4.9. Flow Allocator

The `flowAllocator` module handles (de)allocation request and response calls from the IRM, RIBDaemon, DAFEnrollment or AE.

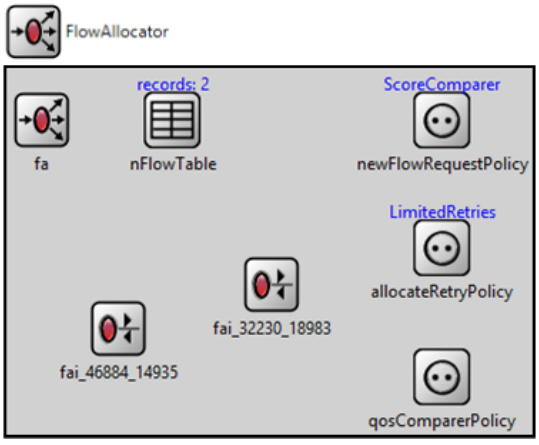

Figure 52. Flow Allocator

5.4.9.1. Submodules

The `flowAllocator` module consists of three submodules (and currently three supported policy interfaces):

- `fa` – core functionality involving instantiation of FAIs;

- `nFlowTable` – mapping between (N)-flow and bound FAI;
- `fai_<portId>_<CEPid>` – managing a whole flow lifecycle.

5.4.9.2. Source codes

Component sources are located in */src/DIF/FA*. It consists of following files:

Filename(s)	**Description**
FA.h/.cc	Implementation of FA core functionality
FABase.h/.cc	Base class for general FA functionality intended for inheritance and extensions
FAI.h/.cc	Connection Table implementation as a table storing state of AP communication
FAIBase.h/.cc	Base class for general FAI functionality intended for inheritance and extensions
FAIListeners.h/cc	FAI Listeners
FAListeners.h/cc	FA listeners
FANotifier.h/.cc	Implementation of FANotifier core functionality
FANotifierBase.h/.cc	Base class for general FANotifier functionality intended for inheritance and extensions
NFlowTable.h/.cc	Interface for NFlowTable entries adding, removing and lookups
NFlowTableEntry.h/.cc	Single Connection Table entry with all its properties
FA.ned	FA simple module
FAI.ned	FA Instance simple module
FANotifier.ned	FANotifier instance for RIBd interaction
FlowAllocator.ned	Flow Allocator compound module holding submodule
NFlowTable.ned	NFlowTable simple module

5.4.9.3. NED design

- FAI and its submodules do not have any gates.

- FAIs are dynamically created and deleted according to the flow lifecycle.
- None of FA submodules has abstract data structures configurable via *config.xml* file.

5.4.9.4. Available policies

Following three policies are associated with FA:

- allocateRetryPolicy - see subchapter 6.2.1[27]
- qosComparePolicy - see subchapter 6.2.2[28]
- newFlowRequestPolicy - see subchapter 6.2.3[29]

5.4.9.5. C++ Implementation

- Registered signals that the Flow Allocator and its submodules are emitting:

```
SIG_FA_MgmtFlowAllocated
SIG_FA_CreateFlowRequestForward
SIG_FA_CreateFlowResponseNegative
SIG_FAI_AllocateRequest
SIG_FAI_DeallocateRequest
SIG_FAI_DeallocateResponse
SIG_FAI_AllocateResponsePositive
SIG_FAI_AllocateResponseNegative
SIG_FAI_CreateFlowRequest
SIG_FAI_DeleteFlowRequest
SIG_FAI_CreateFlowResponsePositive
SIG_FAI_CreateFlowResponseNegative
SIG_FAI_DeleteFlowResponse
```

- Registered signals that the Flow Allocator and its submodules are receiving:

```
SIG_FAI_AllocateResponsePositive
SIG_RIBD_CreateRequestFlow
SIG_ENROLLMENT_Finished
SIG_toFAI_AllocateRequest
SIG_toFAI_AllocateResponseNegative
SIG_AERIBD_AllocateResponsePositive
SIG_RIBD_CreateFlowResponsePositive
```

[27] https://wiki.ict-pristine.eu/wp2/d26/D26-RINASim-Policies-FA-AllocateRetry
[28] https://wiki.ict-pristine.eu/wp2/d26/D26-RINASim-Policies-FA-MultilevelQoS
[29] https://wiki.ict-pristine.eu/wp2/d26/D26-RINASim-Policies-FA-NewFlowRequest

```
SIG_RIBD_CreateFlowResponseNegative
SIG_RIBD_DeleteRequestFlow
SIG_RIBD_DeleteResponseFlow
SIG_RIBD_CreateRequestFlow
```

5.4.9.6. Future work

1. Define interfaces for both FA, FANotifier, FAI;

5.4.10. Relaying and Multiplexing Task

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).

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.

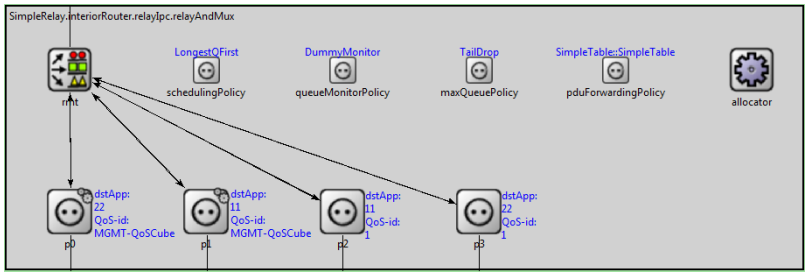

Figure 53. RMT

5.4.10.1. Submodules

`relayAndMux` consists of multiple simple modules of various types, some of which are instantiated only dynamically at runtime.

Static modules:

- `rmt`, the fundamental logic of Relaying And Multiplexing task 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 reconfiguring RMT ports and queues.
- `schedulingPolicy`, the scheduler policy that is invoked by events related to servicing of I/O queues.
- `queueMonitorPolicy`, the monitor policy which is invoked by events related to queue monitoring.
- `maxQueuePolicy`, the policy used for deciding what to do when queue lengths are overflowing their threshold lengths.
- `pduForwardingPolicy`, the policy making the forwarding decisions

Dynamic modules:

- *RMTPort* (encompassed in *RMTPortWrapper*), a representation of one 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 Resource Allocator policies).

5.4.10.2. Source codes

Component sources are located in */src/DIF/RMT*. The folder consists of following files:

Filename(s)	**Description**
RelayAndMux.ned	RMT wrapper (compound module)
RMT.cc/h	implementation of RMT
RMT.ned	RMT simple module
RMTBase.cc/h	abstract class for RMT implementation
RMTModuleAllocator.cc/h	implementation of RMTModuleAllocator
RMTModuleAllocator.ned	RMTModuleAllocator simple module
RMTListeners.cc/h	signal listeners for RMT
RMTPort.cc/h	implementation of RMTPort
RMTPort.ned	RMTPort simple module
RMTPort.ned	a compound wrapper for RMTPort and its RMTQueues
RMTQueue.cc/h	implementation of RMTQueue

Filename(s)	Description
RMTQueue.ned	RMTQueue simple module

5.4.10.3. NED design

`RelayAndMux` parameters:

Parameter	Description
schedPolicyName	module name of desired scheduling policy
qMonitorPolicyName	module name of desired monitor policy
maxQPolicyName	module name of desired maxqueue policy
ForwardingPolicyName	module name of desired PDU forwarding policy
defaultMaxQLength	default maximum queue size
defaultThreshQLength	default threshold queue size
pduTracing	a switch for enabling PDU tracefile generation

5.4.10.4. Available policies

The policies associated with this module are described in subchapter 6.4[30].

5.4.10.5. C++ Implementation

- Registered signals that the RMT module is emitting:
- `RMT-NoConnId` by RMT on received PDU with CEP-id that doesn't match any local EFCP instance
- `RMT-QueuePDURcvd` by a queue on PDU arrival
- `RMT-QueuePDUSent` by a queue on PDU departure
- `RMT-PortPDURcvd` by a port on PDU arrival (coming from a queue)
- `RMT-PortPDUSent` by a port on PDU departure (leaving for an (N-1)-DIF)
- `RMT-PortReadyToServe` by a port when it's ready to serve more PDUs
- `RMT-PortReadyForRead` by a port when it's ready to provide more PDUs
- `RMT-PassUp` to indicate a PDU sent to the (N-1)-flow
- `RMT-PassDown` to indicate a PDU sent from the (N-1)-flow to the RMT

[30] https://wiki.ict-pristine.eu/wp2/d26/D26-RINASim-Policies-RMT

5.4.11. Resource Allocator

The Resource Allocator 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. Its forwarding and queueing functionality are customizable by policies. In RINASim, all the functionality of RA including the policies is encompassed in a single compound module named `resourceAllocator` which is present in every IPC process.

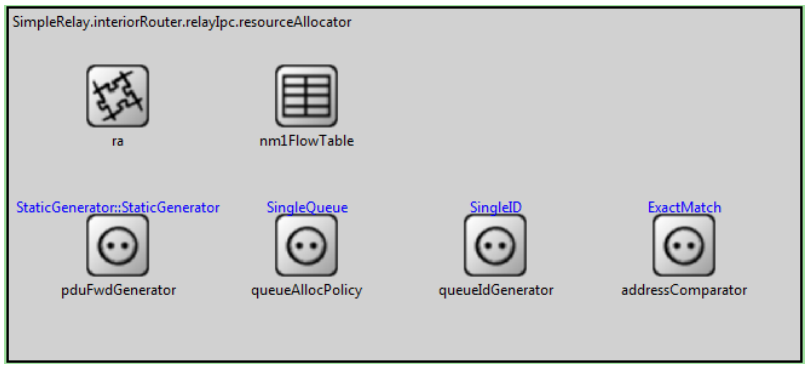

Figure 54. Resource Allocator

5.4.11.1. Submodules

`resourceAllocator` consists of multiple simple modules of various types, namely:

- `ra`, the fundamental logic of Resource Allocator that manages the Relaying and Multiplexing Task and connections to other IPC processes via (N-1)-flows.
- `nm1FlowTable`, a table containing information about the active (N-1)-flows.
- `pduFwdGenerator` (abbreviated PDUFG), a policy which, reacting to various events, 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 that determines whether a PDU address matches the IPC process address.

5.4.11.2. Source codes

Component sources are located in */src/DIF/RA*. The folder consists of following files:

Filename(s)	**Description**
NM1FlowTable.cc/h	implementation of (N-1)-flow table

Filename(s)	Description
NM1FlowTable.ned	(N-1)-flow table simple module
NM1FlowTableItem.cc/h	implementation (N-1)-flow table entry
RA.cc	implementation of RA
RA.ned	RA simple module
RABase.cc/h	abstract class for RA implementation
RAListeners.cc/h	signal listeners for RA
ResourceAllocator.ned	RA wrapper (compound module)

5.4.11.3. NED design

ResourceAllocator parameters:

Parameter	Description
pduftType	module name of the desired PDU Forwarding policy
pdufgPolicyName	module name of the desired PDUFG policy
queueAllocPolicyName	module name of desired QueueAlloc policy
queueIdGenName	module name of desired QueueIDGen policy
addrComparatorName	module name of desired AddrComparator policy

5.4.11.4. Available policies

The policies associated with this module are described in subchapter 6.5[31].

5.4.11.5. C++ Implementation

- Registered signals that the RA module is emitting:

```
RA-CreateFlowPositive
RA-CreateFlowNegative
RA-ExecuteSlowdown
RA-InvokeSlowdown
RA-MgmtFlowAllocated
RA-MgmtFlowDeallocated
```

[31] https://wiki.ict-pristine.eu/wp2/d26/D26-RINASim-Policies-RA

5.4.12. RIB Daemon

The `ribDeamon` is the IPCP's management heart. It receives/sends CDAP management messages and notifies other submodules about management changes.

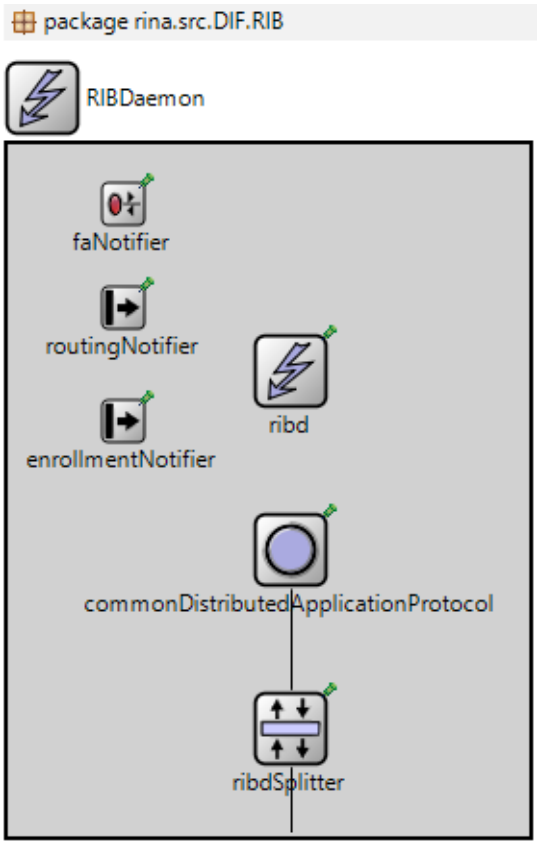

Figure 55. RIB Daemon

5.4.12.1. Submodules

RIBDaemon consists of three submodules:

- `ribd` – core functionality mainly listening to calls from other DIF components and notifying them upon CDAP message reception;
- `commonDistributedApplicationProtocol` – same submodule as in case of DAF components description;
- `ribdSplitter` – splitter is delegating CDAP management messages to/from the `mockEFCPI` or appropriate EFCPIs. Currently, it is just a placeholder forwarding messages between gates.

Other three submodules (`faNotifier`, `routingNotifier`, `enrollmentNotifier`) are notifiers that contain listeners and delegates calls to other IPCP components in case of sending/receiving event to/by RIBd.

5.4.12.2. Source codes

Component sources are located in */src/DIF/RIBD*. It consists of following files:

Filename(s)	Description
RIBd.h/.cc	Implementation of RIBd core functionality
RIBdBase.h/.cc	Base class for general RIBd functionality intended for inheritance and extensions
RIBdListeners.h/cc	RIBd listeners
RIBdSplitter.h/cc	RIBd listeners
RIBd.ned	RIBd processing CDAP messages and delegating them to RA and FA/FAIs
RIBdSplitter.ned	RIBdSplitter simple module forwarding messages
RIBDaemon.ned	Compound module holding all RIBd functionality submodules

5.4.12.3. NED design

- RIBd simulation module design is similar to AE.
- Gates utilized by this submodule:

```
commonDistributedApplicationProtocol.southIo
ribdSplitter.cdapIo
ribdSplitter.rmtIo
ribDaemon.rmtIo;
ribdSplitter.efcpIo
ribDaemon.efcpIo;
```

- None of RIBd submodules has abstract data structures configurable via *config.xml* file.

5.4.12.4. Available policies

No policies are currently associated with this module.

5.4.12.5. C++ Implementation

- Registered signals that RIBd module is emitting:

```
SIG_RIBD_DataSend
SIG_RIBD_CongestionNotification
```

- Registered signals that RIBd module is receiving:

```
SIG_CDAP_DateReceive
SIG_RA_InvokeSlowdown
```

- Registered signals that Notifiers are emitting:

```
SIG_RIBD_CreateRequestFlow
SIG_RIBD_DeleteRequestFlow
SIG_RIBD_DeleteResponseFlow
SIG_AERIBD_AllocateResponsePositive
SIG_AERIBD_AllocateResponseNegative
SIG_RIBD_CreateFlow
SIG_RIBD_CreateFlowResponsePositive
SIG_RIBD_CreateFlowResponseNegative

SIG_RIBD_StartEnrollmentRequest
SIG_RIBD_StartEnrollmentResponse
SIG_RIBD_StopEnrollmentRequest
SIG_RIBD_StopEnrollmentResponse
SIG_RIBD_StartOperationRequest
SIG_RIBD_StartOperationResponse
SIG_RIBD_ConnectionResponsePositive
SIG_RIBD_ConnectionResponseNegative
SIG_RIBD_ConnectionRequest
SIG_RIBD_CACESend

SIG_RIBD_RoutingUpdateReceived
```

- Registered signals that Notifiers are receiving:

```
SIG_FA_CreateFlowRequestForward
SIG_FAI_CreateFlowRequest
SIG_FAI_DeleteFlowRequest
SIG_FAI_DeleteFlowResponse
SIG_FA_CreateFlowResponseNegative
SIG_FAI_CreateFlowResponseNegative
SIG_FAI_CreateFlowResponsePositive
SIG_FA_CreateFlowResponseForward
```

```
SIG_RA_CreateFlowPositive
SIG_RA_CreateFlowNegative
SIG_FAI_AllocateRequest

SIG_CACE_DataReceive
SIG_ENROLLMENT_CACEDataSend
SIG_ENROLLMENT_StartEnrollmentRequest
SIG_ENROLLMENT_StartEnrollmentResponse
SIG_ENROLLMENT_StopEnrollmentRequest
SIG_ENROLLMENT_StopEnrollmentResponse
SIG_ENROLLMENT_StartOperationRequest
SIG_ENROLLMENT_StartOperationResponse

SIG_RIBD_RoutingUpdate
```

5.4.12.6. Future work

1. Probably remove RIBd splitter.

5.4.13. Routing

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

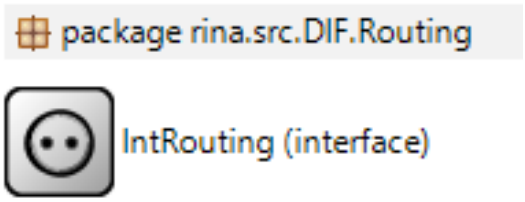

Figure 56. Routing

5.4.13.1. Submodules

None (this is a simple module interface).

5.4.13.2. Source codes

The source codes for each variation of the Routing policy are available in /policies/DIF/Routing.

5.4.13.3. NED design

Policy-specific.

5.4.13.4. Available policies

Routing as a whole is a policy by itself, and there aren't any additional policies to specify.

5.4.13.5. C++ Implementation

Policy-specific.

6. Policies

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 OMNeT NED module interfaces, which helps to minimize the need for modifying existing C/NED source codes. 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 an NED module interface and a base C++ class.

6.1. Used Template

Each atomic RINA component is described using the following set of information:

1. Narrative description when is the policy triggered
2. Existing variants of policy with their brief description
3. Relevant source files containing code of the policy implementation

6.2. Flow Allocator policies

This subchapter discusses the Flow Allocator policies currently supported by RINASim.

6.2.1. AllocateRetry

AllocateRetry occurs whenever initiating FAI receives negative *create flow response*. This policy allows FAI to reformulate the request and/or to recover properly from failure.

6.2.1.1. Variants

- *Base* - Allows unlimited number of retries.
- *LimitedRetries* - Allows retry attempt until `maxCreateFlowRetries` limit is reached.

6.2.1.2. Source codes

Policy source codes are located in */policies/DIF/FA/AllocateRetry* and they contain following files:

Filename(s)	Description
IAllocateRetry.ned	NED interface module
AllocateRetryBase.h/.cc	Base class for general functionality
LimitedRetries/LimitedRetries.h/cc	LimitedRetries version core functionality
LimitedRetries/LimitedRetries.ned	AllocateRetry interface implementation

6.2.2. MultilevelQoS

isValid is invoked during N flow allocation. This policy task consist in check if existing N-1 flows can be used to support an N flow. *setRequirements* is invoked during N flow allocation if no current flow can be used. This policy task consist set the requirements needed for new N-1 flow to support the new N flow.

6.2.2.1. Variants

- *QoSIdComparer* - Only flows with the same QoSId are valid for mapping N to N-1 flows. N flows request same QoS Cube or better to N-1.
- *QoSMinComparer* - Maps N flow to N-1 flows that satisfies N QoS Cube in k equal QoS Cube hops.

6.2.2.2. Source codes

Policy source codes are located in */policies/DIF/FA/MultilevelQoS* and they contain following files:

Filename(s)	Description
IAMultilevelQoS.ned	NED interface module
MultilevelQoS.h/.cc	Base class for general functionality
QoSIdComparerversion /QoSIdComparer.h/cc	QoSIdComparerversion core functionality
QoSIdComparerversion /QoSIdComparer.ned	NewFlowRequest interface implementation

Filename(s)	Description
QoSMinComparer/QoSMinComparer.h/cc	QoSMinComparerversion core functionality
QoSMinComparer/QoSMinComparer.ned	QoSMinComparerinterface implementation

6.2.3. NewFlowRequest

NewFlowRequst 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.

6.2.3.1. Variants

- *Base* - Implicitly accepts any new flow.
- *MinComparer* - The first QoSCube meeting ALL QoS requirements is chosen.
- *ScoreComparer* - Mapped QoSCube is chosen based on computed score. The score is incremented for each QoSCube parameter meeting QoS requirement. Otherwise, the score is decremented.

6.2.3.2. Source codes

Policy source codes are located in */policies/DIF/FA/NewFlowRequest* and they contain following files:

Filename(s)	Description
INewFlowRequest.ned	NED interface module
NewFlowRequestBase.h/.cc	Base class for general functionality
MinComparer/MinComparer.h/cc	MinComparer version core functionality
MinComparer/MinComparer.ned	NewFlowRequest interface implementation
ScoreComparer/ScoreComparer.h/cc	ScoreComparer version core functionality
ScoreComparer/ScoreComparer.ned	NewFlowRequest interface implementation

6.3. EFCP policies

This subchapter discusses EFCP policies. Policies are further structured into two subsections: DTP and DTCP. All DTP and DTCP policies share the same base class `EFCPPolicy` that provides a common interface-like approach to their invocation. Upon invocation, each policy is

provided with `DTPState` and `DTCPState` objects. Each *Base policy class implements the default action that is executed if the specified policy returns `true` or if no policy is specified.

As an example, all default actions are also created as a standalone policy with name `<policyName>PolicyDefault` which directly executes parent `defaultAction` method and then returns `false` to prevent calling the default action again. Among all known policies from specification belong:

DTP:

- `InitialSeqNumPolicy`[32] - This policy allows some discretion in selecting the initial sequence number when DRF is going to be sent.
- `RcvrInactivityPolicy`[33] - If no PDUs arrive in this period, the receiver should expect a DRF in the next Transfer PDU. If not, something is very wrong. It should generally be set to 2(MPL+R+A).
- `SenderInactivityPolicy`[34] - This policy is used to detect long periods of no traffic, indicating that a DRF should be sent. If not, something is very wrong. It should generally be set to 3(MPL+R+A).
- `DTPRTTEstimatorPolicy`[35] - This policy is executed by the sender to estimate the duration of the retransmission timer. This policy will be based on an estimate of round-trip time and the Ack or Ack List policy in use.
- `UnknownFlow` - When a PDU arrives for a Data Transfer Flow terminating in this IPC-Process and there is no active DTSV, this policy consults the ResourceAllocator to determine what to do.

DTCP:

- `ECNPolicy`[36] - This policy is invoked upon receiving PDU with ECN set in the header.
- `ECNSlowDownPolicy`[37] - This policy is invoked upon RA receives the SlowDown request from relaying node.
- `LostControlPDUPolicy`[38] - This policy determines what action to take when the PM detects that a control PDU (Ack or Flow Control) may have been lost. If this procedure

[32] D26-RINASim-Policies-EFCP-InitialSequenceNumber
[33] D26-RINASim-Policies-EFCP-RcvrTimerInactivity
[34] D26-RINASim-Policies-EFCP-SenderTimerInactivity
[35] D26-RINASim-Policies-EFCP-RTTEstimator
[36] D26-RINASim-Policies-EFCP-ECN
[37] D26-RINASim-Policies-EFCP-ECNSlowDown
[38] D26-RINASim-Policies-EFCP-LostControlPDU

returns True, then the PM will send a Control-Ack and an empty Transfer PDU. If it returns False, then any action is determined by the policy.

- `NoOverridePeakPolicy`[39] - This policy allows rate-based flow control to exceed its nominal rate. Presumably this would be for short periods, and policies should enforce this. Like all policies, if this returns True it creates the default action that is no override.
- `NoRateSlowDownPolicy`[40] - This policy is used to lower momentarily the send rate below the rate allowed.
- `RateReductionPolicy`[41] - This policy is executed in case of Rate-based Flow Control and if a condition of local shortage of buffers occurs or when the condition is opposite and buffers are less full than a given threshold so that rate can be increased to the rate agreed during the connection establishment.
- `RcvFCOverrunPolicy`[42] - This policy determines what action to take if the receiver receives PDUs, but the credit or rate has been exceeded. If this procedure returns True, then the PDU is discarded; otherwise PDU processing is allowed to continue normally.
- `RcvrAckPolicy`[43] - This policy is executed by the receiver of the PDU and provides some discretion in the action taken. The default action is to either Ack immediately or to start the A-Timer and Ack the LeftWindowEdge when it expires.
- `RcvrControlAckPolicy`[44] - This policy is executed by the receiver of Control-Ack PDU. Its purpose is to recover faster from PM inconsistency.
- `RcvrFCPolicy`[45] - This policy is invoked when a Transfer PDU is received to give the receiving PM an opportunity to update the flow control allocations.
- `ReceivingFCPolicy`[46] - This policy is invoked by the receiver of PDU in case there is a Flow Control present, but no Retransmission Control. The default action is to send FlowControl PDU.
- `ReconcileFCPolicy`[47] - This policy is invoked when both Credit and Rate-based flow control are in use, and they disagree on whether the PM can send or receive data. If it returns True, then the PM can send or receive; if False, it cannot.

[39] D26-RINASim-Policies-EFCP-NoOverrideDefaultPeak
[40] D26-RINASim-Policies-EFCP-NoRate-SlowDown
[41] D26-RINASim-Policies-EFCP-RateReduction
[42] D26-RINASim-Policies-EFCP-RcvFlowControlOverrun
[43] D26-RINASim-Policies-EFCP-RcvrAck
[44] D26-RINASim-Policies-EFCP-RcvrControlAck
[45] D26-RINASim-Policies-EFCP-RcvrFlowControl
[46] D26-RINASim-Policies-EFCP-ReceivingFlowControl
[47] D26-RINASim-Policies-EFCP-ReconcileFlowConflict

- `RxTimerExpiryPolicy`[48] - This policy is executed by the sender when a Retransmission Timer Expires. If this policy returns True, then all PDUs with the sequence number less than or equal to the sequence number of the PDU associated with this timeout are retransmitted; otherwise the procedure must determine what action to take. This policy must be executed in less than the maximum time to Ack.
- `SenderAckPolicy`[49] - This policy is executed by the Sender and provides the Sender with some discretion on when PDUs may be deleted from the ReTransmissionQ. This is useful for multicast and similar situations where one might want to delay discarding PDUs from the retransmission queue.
- `SenderAckListPolicy`[50] - This policy is executed by the Sender and provides the Sender with some discretion on when PDUs may be deleted from the ReTransmissionQ. This policy is used in conjunction with the selective acknowledgement aspects of the mechanism. This is useful for multicast and similar situations where one might want to delay discarding PDUs from the retransmission queue.
- `SendingAckPolicy`[51] - This policy is executed upon A-Timer expiration in case there is DTCP present. The default action is to update Receiver Left Window Edge, invoke delimiting and to send Ack/FlowControl PDU.
- `SndFCOverrunPolicy`[52] - This policy determines what action to take if the receiver receives PDUs, but the credit or rate has been exceeded. If this procedure returns True, then the PDU is discarded; otherwise PDU processing is allowed to continue normally.
- `TxControlPolicy`[53] - This policy is used when there are conditions that warrant sending fewer PDUs than allowed by the sliding window flow control, e.g. the ECN bit is set.

From all of the mentioned policies, RINASim does not support only `UnknownFlow` policy, but it performs its default action i.e. deletes incoming PDU that does not belong to any known flow.

6.3.1. DTP: InitialSequenceNumber

InitialSeqNum policy allows some discretion in selecting the initial sequence number when DRF is going to be sent.

[48] D26-RINASim-Policies-EFCP-RetransmissionTimerExpiry
[49] D26-RINASim-Policies-EFCP-SenderAck
[50] D26-RINASim-Policies-EFCP-SenderAckList
[51] D26-RINASim-Policies-EFCP-SendingAck
[52] D26-RINASim-Policies-EFCP-SndFlowControlOverrun
[53] D26-RINASim-Policies-EFCP-TransmissionControl

6.3.1.1. Variants

- *Default* - Default actions set the new seq num to 1.

6.3.1.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTP/InitialSeqNum* and they contain following files:

Filename(s)	Description
IntInitialSeqNumPolicy.ned	NED interface module
InitialSeqNumPolicyBase.h/.cc	Base class for general functionality
InitialSeqNumPolicyDefault/ InitialSeqNumPolicyDefault.ned	Simple module
InitialSeqNumPolicyDefault/ InitialSeqNumPolicyDefault.h/.cc	Policy invoking only default action

6.3.2. DTP: RTTEstimator

RTTEstimator policy is executed by the sender to estimate the duration of the retransmission timer. This policy will be based on an estimate of round-trip time and the Ack or Ack List policy in use.

6.3.2.1. Variants

- *Default* - Computes Round trip time only as an average from current and the last computed RTT.

6.3.2.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTP/RTTEstimator* and they contain following files:

Filename(s)	Description
IntRTTEstimatorPolicy.ned	NED interface module
RTTEstimatorPolicyBase.h/.cc	Base class for general functionality
RTTEstimatorPolicyDefault/ RTTEstimatorPolicyDefault.ned	Simple module
RTTEstimatorPolicyDefault/ RTTEstimatorPolicyDefault.h/.cc	Base class for general functionality

6.3.3. DTP: RcvrTimerInactivity

RcvrTimerInactivity policy is used when DTCP is in use. If no PDUs arrive in this period, the receiver should expect a DRF in the next Transfer PDU. If not, something is very wrong. It should be set to 2(MPL+R+A).

6.3.3.1. Variants

- *Default* - Resets all receiver-side variables and queues.

6.3.3.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTP/RcvrTimerInactivity* and they contain following files:

Filename(s)	**Description**
IntRcvrTimerInactivityPolicy.ned	NED interface module
IntRcvrTimerInactivityPolicyBase.h/.cc	Base class for general functionality
RcvrTimerInactivityPolicyDefault/ RcvrTimerInactivityPolicyDefault.ned	Simple module
RcvrTimerInactivityPolicyDefault/ RcvrTimerInactivityPolicyDefault.h/.cc	Policy invoking only default action

6.3.4. DTP: SenderInactivityTimer

SenderInactivityTimer policy is used when DTCP is in use. This timer is used to detect long periods of no traffic, indicating that a DRF should be sent. If not, something is very wrong. It should be set to 3(MPL+R+A).

6.3.4.1. Variants

- *Default* - Resets all sender-side variables and queues.

6.3.4.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTP/SenderInactivityTimer* and they contain following files:

Filename(s)	**Description**
IntSenderInactivityTimerPolicy.ned	NED interface module
SenderInactivityTimerPolicyBase.h/.cc	Base class for general functionality

Filename(s)	Description
SenderInactivityTimerPolicyDefault/ SenderInactivityTimerPolicyDefault.ned	Simple module
SenderInactivityTimerPolicyDefault/ SenderInactivityTimerPolicyDefault.h/.cc	Policy invoking only default action

6.3.5. DTCP: ECN

ECN policy handles ECN bit in incoming DT-PDUs.

6.3.5.1. Variants

- *Default* - Sets inner variable based on bit in DT-PDU header.

6.3.5.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/ECN* and they contain following files:

Filename(s)	Description
IntECNPolicy.ned	NED interface module
ECNPolicyBase.h/.cc	Base class for general functionality
ECNPolicyDefault/ECNPolicyDefault.ned	Simple module
ECNPolicyDefault/ECNPolicyDefault.h/.cc	Policy invoking only default action

6.3.6. DTCP: ECNSlowDown

ECNSlowDown policy is executed upon IPCP's RA receives Congestion Notification.

6.3.6.1. Variants

- *Default* - Default action is empty.

6.3.6.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/ECNSlowDown* and they contain following files:

Filename(s)	Description
IntECNlowDownPolicy.ned	NED interface module

Filename(s)	Description
ECNlowDownPolicyBase.h/.cc	Base class for general functionality
ECNlowDownPolicyDefault/ ECNlowDownPolicyDefault.ned	Simple module
ECNlowDownPolicyDefault/ ECNlowDownPolicyDefault.h/.cc	Policy invoking only default action

6.3.7. DTCP: LostControlPDU

LostControlPDU policy determines what action to take when the PM detects that a control PDU (Ack or Flow Control) may have been lost. If this procedure returns True, then the PM will send a Control-Ack and an empty Transfer PDU. If it returns False, then any action is determined by the policy.

6.3.7.1. Variants

- *Default* - Sends ControlAck and empty DT-PDU.

6.3.7.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/LostControlPDU* and they contain following files:

Filename(s)	Description
IntLostControlPDUPolicy.ned	NED interface module
LostControlPDUPolicyBase.h/.cc	Base class for general functionality
LostControlPDUPolicyDefault/ LostControlPDUPolicyDefault.ned	Simple module
LostControlPDUPolicyDefault/ LostControlPDUPolicyDefault.h/.cc	Policy invoking only default action

6.3.8. DTCP: NoOverridePeak

NoOverridePeak policy allows rate-based flow control to exceed its nominal rate. Presumably this would be for short periods, and policies should enforce this. Like all policies, if this returns True it creates the default action that is no override.

6.3.8.1. Variants

- *Default* - Puts DT-PDU on ClosedWindowQ.

6.3.8.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/NoOverridePeak* and they contain following files:

Filename(s)	Description
IntNoOverridePeakPolicy.ned	NED interface module
NoOverridePeakPolicyBase.h/.cc	Base class for general functionality
NoOverridePeakPolicyDefault/ NoOverridePeakPolicyDefault.ned	Simple module
NoOverridePeakPolicyDefault/ NoOverridePeakPolicyDefault.h/.cc	Policy invoking only default action

6.3.9. DTCP: NoRateSlowDown

NoRate-SlowDown policy is used to lower momentarily the send rate below the rate allowed.

6.3.9.1. Variants

- *Default* - Default action does not slow down.

6.3.9.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/NoRateSlowDown* and they contain following files:

Filename(s)	Description
IntNoRateSlowDownPolicy.ned	NED interface module
NoRateSlowDownPolicyBase.h/.cc	Base class for general functionality
NoRateSlowDownPolicyDefault/ NoRateSlowDownPolicyDefault.ned	Simple module
NoRateSlowDownPolicyDefault/ NoRateSlowDownPolicyDefault.h/.cc	Policy invoking only default action

6.3.10. DTCP: RateReduction

RateReduction policy is executed in case of Rate-based Flow Control and if a condition of local shortage of buffers occurs or when the condition is opposite and buffers are less full than a given threshold so that rate can be increased to the rate agreed during the connection establishment.

6.3.10.1. Variants

- *Default* - Slow down 10% if buffers are getting low.

6.3.10.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/RateReduction* and they contain following files:

Filename(s)	Description
IntRateReductionPolicy.ned	NED interface module
RateReductionPolicyBase.h/.cc	Base class for general functionality
RateReductionPolicyDefault/ RateReductionPolicyDefault.ned	Simple module
RateReductionPolicyDefault/ RateReductionPolicyDefault.h/.cc	Policy invoking only default action

6.3.11. DTCP: RcvFlowControlOverrun

RcvFlowControlOverrun This policy determines what action to take if the receiver receives PDUs, but the credit or rate has been exceeded. If this procedure returns True, then the PDU is discarded; otherwise PDU processing is allowed to continue normally.

6.3.11.1. Variants

- *Default* - Default action is to drop the PDU and to send control PDU.

6.3.11.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/RcvFCOverrun* and they contain following files:

Filename(s)	Description
IntRcvFCOverrunPolicy.ned	NED interface module
RcvFCOverrunPolicyBase.h/.cc	Base class for general functionality
RcvFCOverrunPolicyDefault/ RcvFCOverrunPolicyDefault.ned	Simple module
RcvFCOverrunPolicyDefault/ RcvFCOverrunPolicyDefault.h/.cc	Policy invoking only default action

6.3.12. DTCP: RcvrAck

RcvrAck policy is executed by the receiver of the DT-PDU and provides some discretion in the action taken. The default action is to either Ack immediately or to start the A-Timer and Ack the RcvLeftWindowEdge when it expires.

6.3.12.1. Variants

- *Default* - Sends Ack.

6.3.12.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/RcvrAck* and they contain following files:

Filename(s)	**Description**
IntRcvrAckPolicy.ned	NED interface module
RcvrAckPolicyBase.h/.cc	Base class for general functionality
RcvrAckPolicyDefault/ RcvrAckPolicyDefault.ned	Simple module
RcvrAckPolicyDefault/ RcvrAckPolicyDefault.h/.cc	Policy invoking only default action

6.3.13. DTCP: RcvrControlACK

RcvrControlAck policy is executed upon reception of ControlAck PDU.

6.3.13.1. Variants

- *Default* - Default action is to check the values and if necessary send back Control PDU with updated information.

6.3.13.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/RcvrControlAck* and they contain following files:

Filename(s)	**Description**
IntRcvrControlAckPolicy.ned	NED interface module

Filename(s)	Description
RcvrControlAckPolicyBase.h/.cc	Base class for general functionality
RcvrControlAckPolicyDefault/ RcvrControlAckPolicyDefault.ned	Simple module
RcvrControlAckPolicyDefault/ RcvrControlAckPolicyDefault.h/.cc	Policy invoking only default action

6.3.14. DTCP: RcvrFlowControl

RcvrFlowControl policy is invoked when a Transfer PDU is received to give the receiving PM an opportunity to update the flow control allocations.

6.3.14.1. Variants

- *Default* - Increment RRWE.

6.3.14.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/RcvrFC* and they contain following files:

Filename(s)	Description
RcvrFlowControl.ned	NED interface module
IntRcvrFCPolicy.ned	NED interface module
RcvrFClPolicyBase.h/.cc	Base class for general functionality
RcvrFCPolicyDefault/ RcvrFCPolicyDefault.ned	Simple module
RcvrFCPolicyDefault/ RcvrFCPolicyDefault.h/.cc	Policy invoking only default action

6.3.15. DTCP: ReceivingFlowControl

ReceivingFlowControl policy is invoked by the receiver of a DataTransferPDU in case there is a Flow Control present, but no Retransmission Control. The default action is to send FlowControlPDU.

6.3.15.1. Variants

- *Default* - Send Control PDU with Flow Control Informations.

6.3.15.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/ReceivingFC* and they contain following files:

Filename(s)	Description
ReceivingFC.ned	NED interface module
IntReceivingFCPolicy.ned	NED interface module
ReceivingFCPolicyBase.h/.cc	Base class for general functionality
ReceivingFCPolicyDefault/ ReceivingFCPolicyDefault.ned	Simple module
ReceivingFCPolicyDefault/ ReceivingFCPolicyDefault.h/.cc	Policy invoking only default action

6.3.16. DTCP: ReconcileFlowConflict

ReconcileFlowConflict policy is invoked when both Credit and Rate-based flow control are in use, and they disagree on whether the PM can send or receive data. If it returns True, then the PM can send or receive; if False, it cannot.

6.3.16.1. Variants

- *Default* - Does nothing.

6.3.16.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/ReconcileFC* and they contain following files:

Filename(s)	Description
IntReconcileFCPolicy.ned	NED interface module
ReconcileFCPolicyBase.h/.cc	Base class for general functionality
ReconcileFCPolicyDefault/ ReconcileFCPolicyDefault.ned	Simple module
ReconcileFCPolicyDefault/ ReconcileFCPolicyDefault.h/.cc	Policy invoking only default action

6.3.17. DTCP: RetransmissionTimerExpiry

RetransmissionTimerExpiry policy is executed by the sender when a Retransmission Timer Expires. If this policy returns True, then all PDUs with the sequence number less than or equal to the sequence number of the PDU associated with this timeout are retransmitted; otherwise the procedure must determine what action to take. This policy must be executed in less than the maximum time to Ack.

6.3.17.1. Variants

- *Default* - Retransmits PDU with seq num equal to the one in RXTimer.

6.3.17.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/RxTimerExpiry* and they contain following files:

Filename(s)	**Description**
IntRxTimerExpiryPolicy.ned	NED interface module
RxTimerExpiryPolicyBase.h/.cc	Base class for general functionality
RxTimerExpiryPolicyDefault/ RxTimerExpiryPolicyDefault.ned	Simple module
RxTimerExpiryPolicyDefault/ RxTimerExpiryPolicyDefault.h/.cc	Policy invoking only default action

6.3.18. DTCP: SenderAck

SenderAck policy is executed by the Sender and provides the Sender with some discretion on when PDUs may be deleted from the RetransmissionQ. This is useful for multicast and similar situations where one might want to delay discarding PDUs from the retransmission queue.

6.3.18.1. Variants

- *Default* - Removes DT-PDU from Retransmission Queue up to Acked sequence number.

6.3.18.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/SenderAck* and they contain following files:

Filename(s)	Description
IntSenderAckPolicy.ned	NED interface module
SenderAckPolicyBase.h/.cc	Base class for general functionality
SenderAckPolicyDefault/ SenderAckPolicyDefault.ned	Simple module
SenderAckPolicyDefault/ SenderAckPolicyDefault.h/.cc	Policy invoking only default action

6.3.19. DTCP: SenderAckList

SenderAckList policy is executed by the Sender and provides the Sender with some discretion on when PDUs may be deleted from the ReTransmissionQ. This policy is used in conjunction with the selective acknowledgement aspects of the mechanism. This is useful for multicast and similar situations where to want to delay discarding PDUs from the retransmission queue.

6.3.19.1. Variants

- *Default* - Removes specified seq num ranges from from Retransmission Queue.

6.3.19.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/SenderAckList* and they contain following files:

Filename(s)	Description
IntSenderAckListPolicy.ned	NED interface module
SenderAckListPolicyBase.h/.cc	Base class for general functionality
SenderAckListPolicyDefault/ SenderAckListPolicyDefault.ned	Simple module
SenderAckListPolicyDefault/ SenderAckListPolicyDefault.h/.cc	Policy invoking only default action

6.3.20. DTCP: SendingAck

SendingAck policy is executed upon A-Timer expiration in case there is DTCP present. The default action is to update RcvLeftWindowEdge, invoke delimiting and to send Ack/ FlowControlPDU.

6.3.20.1. Variants

- *Default* - Updates RcvLeftWindowEdge, invokes delimiting and sends Ack/FlowControlPDU.

6.3.20.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/SendingAck* and they contain following files:

Filename(s)	Description
IntSendingAckPolicy.ned	NED interface module
SendingAckPolicyBase.h/.cc	Base class for general functionality
SendingAckPolicyDefault/ SendingAckPolicyDefault.ned	Simple module
SendingAckPolicyDefault/ SendingAckPolicyDefault.h/.cc	Policy invoking only default action

6.3.21. DTCP: SndFlowControlOverrun

SndFlowControlOverrun - policy determines what action to take if the Sender has PDU to send but its SndRightWindowEdge or SndRate prevents him from sending it. The default action is to put it in ClosedWindowQueue.

6.3.21.1. Variants

- *Default* - Puts DT-PDU in ClosedWindowQueue.

6.3.21.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/SndFCOverrun* and they contain following files:

Filename(s)	Description
IntSndFCOverrunPolicy.ned	NED interface module
SndFCOverrunPolicyBase.h/.cc	Base class for general functionality
SndFCOverrunPolicyDefault/ SndFCOverrunPolicyDefault.ned	Simple module
SndFCOverrunPolicyDefault/ SndFCOverrunPolicyDefault.h/.cc	Policy invoking only default action

6.3.22. DTCP: Transmission Control

TransmissionControl policy is used when there are conditions that warrant sending fewer PDUs than allowed by the sliding window flow control.

6.3.22.1. Variants

- *Default* - Puts as many DT-PDUs from `generatedPDUQ` to `postablePDUQ`.

6.3.22.2. Source codes

Policy source codes are located in */policies/DIF/EFCP/DTCP/TXControl* and they contain following files:

Filename(s)	**Description**
IntTXControlPolicy.ned	NED interface module
TXControlPolicyBase.h/.cc	Base class for general functionality
TXControlPolicyDefault/ TXControlPolicyDefault.ned	Simple module
TXControlPolicyDefault/ TXControlPolicyDefault.h/.cc	Policy invoking only default action

6.4. Resource Allocator Policies

This subchapter discusses RA policies. Since there currently aren't any available specifications for policies needed in Resource Allocator, all following policies are simulator-specific.

- *AddressComparator*
- *PDUForwardingGenerator*
- *QueueAlloc*
- *QueueIDGen*

6.4.1. AddressComparator

AddressComparator is invoked by RMT and its policies to determine whether a PDU address matches the IPC process address. This is used mainly on message arrival to decide whether the PDU is directed to the IPC process.

6.4.1.1. Variants

- *ExactMatch* - Provides exact matching.
- *PrefixMatch* - Provides matching based on common prefix.

6.4.1.2. Source codes

Policy source codes are located in */policies/DIF/RA/AddressComparator* and they contain following files:

Filename(s)	Description
IntAddressComparator.ned	NED interface module
AddressComparatorBase.cc/h	Base class for general functionality
ExactMatch/ExactMatch.cc/h	ExactMatch version core functionality
ExactMatch/ExactMatch.ned	ExactMatch interface implementation
PrefixMatch/PrefixMatch.cc/h	PrefixMatch version core functionality
PrefixMatch/PrefixMatch.ned	PrefixMatch interface implementation

6.4.2. PDU Forwarding Generator

PDUFG (PDU Forwarding Generator) manages the PDU Forwarding policy (traditionally a forwarding table), usually by means of adding and removing forwarding information. For this purpose, PDUFG uses pieces of information provided by other sources, most notably the Routing policy.

6.4.2.1. Variants

- *StaticGenerator* - The default implementation using forwarding information statically provided to DIF Allocator via XML configuration.
- *SimpleGenerator* - The simplest dynamic generator proxying information provided by the routing policy.
- *HierarchicalGenerator* - An implementation working with hierarchical addresses (e.g. A.B.C.…)
- *HopsSingle1Entry* - An implementation for hop-based routing managing only one port per destination address.
- *HopsSingleMEntries* - An implementation for hop-based routing managing multiple ports per destination address.

- *LatencySingle1Entry* - An implementation for latency-based routing managing only one port per destination address.
- *LatencySingleMEntries* - An implementation for latency-based routing managing multiple ports per destination address.
- *SingleDomainGenerator* - Domain routing: generator with a single domain
- *BiDomainGenerator* - Domain routing: generator with two domains per IPC process
- *QoSDomainGenerator* - Domain routing: QoS-based generator

6.4.3. QueueAlloc

QueueAlloc policy manages allocation and deallocation of RMT queues in response to events happening inside the IPC process. This allows for flexibility when experimenting with queuing disciplines.

6.4.3.1. Variants

- *SingleQueue* - A pair of queues per (N-1)-port.
- *QueuePerNFlow* - A pair of queues per active (N-1)-flow + a pair of management queues.
- *QueuePerNQoS* - A pair of queues per each available QoS cube.
- *QueuePerNCU* - A pair of queues per each available Cherish/Urgency class. QoS Cubes are mapped each to a Cherish/Urgency class.

6.4.4. PDU Forwarding Generator

PDUFG (PDU Forwarding Generator) manages the PDU Forwarding policy (traditionally a forwarding table), usually by means of adding and removing forwarding information. For this purpose, PDUFG uses pieces of information provided by other sources, most notably the Routing policy.

6.4.4.1. Variants

- *StaticGenerator* - The default implementation using forwarding information statically provided to DIF Allocator via XML configuration.
- *SimpleGenerator* - The simplest dynamic generator proxying information provided by the routing policy.
- *HierarchicalGenerator* - An implementation working with hierarchical addresses (e.g. A.B.C.…)

- *HopsSingle1Entry* - An implementation for hop-based routing managing only one port per destination address.
- *HopsSingleMEntries* - An implementation for hop-based routing managing multiple ports per destination address.
- *LatencySingle1Entry* - An implementation for latency-based routing managing only one port per destination address.
- *LatencySingleMEntries* - An implementation for latency-based routing managing multiple ports per destination address.
- *SingleDomainGenerator* - Domain routing: generator with a single domain
- *BiDomainGenerator* - Domain routing: generator with two domains per IPC process
- *QoSDomainGenerator* - Domain routing: QoS-based generator

6.4.5. QueueIDGen

QueueIDGen is a companion policy to the *QueueAlloc* policy and provides generation of queue IDs from given objects (PDUs/flow specifics).

6.4.5.1. Variants

- *SingleQueue* - Returns "0".
- *QueuePerNFlow* - Returns a concatenation of the other endpoint's IPC address and CEP-id.
- *QueuePerNQoS* - Returns a QoS-cube ID.
- *QueuePerNCU* - Returns a Cherish/Urgency class of the QoS-cube ID. BE if not defined.

6.5. RMT Policies

This subchapter discusses RMT policies. According to the specifications, RMT providies three policies:

- *Scheduler*
- *Monitor*
- *MaxQueue*

RINASim RMT implements those policies and additionally contains one RINASim-specific policy:

- *PDUForwarding*

6.5.1. MaxQueue

MaxQueue is a policy used for deciding what to do when queue lengths are overflowing their threshold lengths. It's invoked whenever the size of items in a queue reaches above a threshold.

6.5.1.1. Variants

- *TailDrop* - A policy that drops arriving PDUs when queue size >= allowed maximum.
- *ECNMarker* - A policy that marks arriving PDUs when queue size >= threshold and drops them when queue size >= allowed maximum.
- *ReadRateReducer* - A policy that causes RMT to stop receiving data from relevant (N-1)-ports when queue size >= allowed maximum.
- *UpstreamNotifier* - A policy that causes a notification to be sent to the PDU sender when queue size >= allowed maximum.
- *REDDropper* - A policy used for for the Random Early Detection feature.
- *DumbMaxQ* - A policy used in conjunction with Monitor policies extending the SmartMonitor interface. Drop a new PDU heuristically depending on a probability given by the monitor.

6.5.2. Monitor

Monitor is a stateful policy that manages variables used by other RMT policies. It's invoked by various events happening inside RMT and its ports and queues.

6.5.2.1. Variants

- *DummyMonitor* - A noop implementation.
- *REDMonitor* - A monitor used for for the Random Early Detection feature.
- *SmartMonitor* - A monitor interface that joins all scheduling related tasks (monitor, maxQ and scheduling)
- *WeightedFairQMonitor* - A monitor used to compute rates for WFW.
- *BEMonitor* - Extends SmartMonitor. Implementation of Best-effort.
- *DLMonitor* - Extends SmartMonitor. Implementation of Cherish/Urgency monitor.
- *DQMonitor* - Extends SmartMonitor. Implementation of DeltaQ monitor.
- *eDLMonitor* - Extends SmartMonitor. Implementation of an enhanced version of Cherish/Urgency monitor.

6.5.3. PDUForwarding

PDUForwarding 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 .

6.5.3.1. Variants

- *SimpleTable* - A table with {(dstAddr, QoS) $\rightarrow$ port} mappings.
- *MiniTable* - A table with {dstAddr $\rightarrow$ port} mappings.
- *MultiMiniTable* - A table with {dstAddr $\rightarrow$ vector<port>} mappings.
- *FloodMiniTable* - A table with {dstAddr $\rightarrow$ port} mappings.
- *DomainTable* - Two tables {(prefix, QoS) $\rightarrow$ domainID} {(domainID, address) $\rightarrow$ port}} mappings.
- *HierarchicalTable* - A table with {prefix $\rightarrow$ {(infix, QoS) $\rightarrow$ port}} mappings.
- *QoSTable* - A table with {(dstAddr, QoS) $\rightarrow$ port} mappings.

6.5.4. Scheduler

Scheduler is invoked each time some (N-1)-port has data to send and uses an algorithm to make a decision about which of port's queues should be handled first.

6.5.4.1. Variants

- *LongestQFirst* - Always picks the queue which contains the most PDUs.
- *DumbSch* - A policy used in conjunction with Monitor policies extending the SmartMonitor interface. Returns the PDU decided by the monitor.
- *WeightedFairQ* - Picks the queues depending on datarate of QoS
- *DQSch* - A policy used in conjunction with DQMonitor. Returns the PDU decided by the monitor or waits some time to space bursts.

6.6. Routing policies

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

6.6.1. Variants

- *DummyRouting* - Does nothing.
- *DomainRouting* - Exchanges routing information of distinct routing domains configured either with link-state or distance-vector (see subchapter 7.3.3[54] for detailed description).
- *SimpleRouting* - Exchanges routing information of distinct routing domains based on QoS Cube configured either with link-state (see subchapter 7.3.1[55]) or distance-vector (see subchapter 7.3.2[56]).
- *TDomainRouting* - Exchanges routing information of distinct routing domains configured either with link-state or distance-vector. Metric data-type is defined by template.
- *TSimpleRouting* - Exchanges routing information of distinct routing domains based on QoS Cube configured either with link-state or distance-vector. Metric data-type is defined by template.

[54] D26-RINASim-PolicyFeatures-Routing-Domain
[55] D26-RINASim-PolicyFeatures-Routing-TSimpleLS
[56] D26-RINASim-PolicyFeatures-Routing-TSimpleDV

7. Policy-driven Features

7.1. Congestion Avoidance

7.1.1. Legacy Random Early Detection

Port of the legacy Random Early Detection algorithm, included mainly for demonstrating RINA's programmability.

7.1.1.1. Policy set

- */policies/DIF/RMT/Monitor/REDMonitor*
- */policies/DIF/RMT/MaxQueue/REDDropper*

7.1.1.2. Configuration

Module	Variable	Description	Default value
REDDropper	double dropProbability	probability of packet drop	0.4
REDDropper	bool marking	applies ECN markings on PDUs instead of dropping them	false

7.1.1.3. References

- [RED]

7.1.2. TCP-like congestion avoidance

A set of simple TCP-like congestion control policies is specified here to demonstrate RINA's congestion control capabilities. The *TxControlPolicyTCPTahoe* policy is an implementation of *TxControl* policy of DTCP. It defines a set of internal variables such as congestion window size (CWND) to control the number of sent packets additionally based on congestion signals such as ECN and pushback. Therefore, the given credit to the sender is the minimum of the one allowed by CWND and the flow control's window. The congestion controller of this policy behaves similarly to the one in TCP [RFC5681].

Since TCP's congestion controller is coupled with other functions such as round-trip-time (RTT) and retransmission timeout (RTO) estimations and acknowledgement packets treatment,

two other policies, called *RTTEstimatorPolicyTCP* and *SenderAckPolicyTCPTahoe* have been additionally defined to respectively handle those functions in RINA. *RTTEstimatorPolicyTCP* calculates RTT and RTO based on the method presented in RFC 6298 [RFC6298].

7.1.2.1. Policy set

- */policies/DIF/EFCP/DTCP/TxControl/TxControlPolicyTCPTahoe*
- */policies/DIF/EFCP/DTCP/SenderAck/SenderAckPolicyTCPTahoe*
- */policies/DIF/EFCP/DTP/RTTEstimator/RTTEstimatorPolicyTCP*

7.1.2.2. Configuration

Module	Variable	Description	Default value
TxControlPolicyTCPTahoe	int packetSize	packet size	536

7.2. Scheduling

7.2.1. Delay-loss

Delay Loss scheduling based on strict cherish/urgency distinction.

7.2.1.1. Policy set

- */policies/DIF/RMT/Monitor/DLMonitor*
- */policies/DIF/RMT/MaxQueue/DumbMaxQ*
- */policies/DIF/RMT/Scheduler/DumbSch*

7.2.1.2. Configuration

Module	Variable	Description	Default value
DLMonitor	xml cuData (array CUItem)	Cherish/Urgency classes definition	empty xml

CUItem - Cherish/Urgency Class definition

Parameter	Type	DataType	Description
id	attribute	string	Name of the C/U class
queue	element	string	Queue of the C/U class

Parameter	Type	DataType	Description
urgency	element	int	Priority of the C/U class
cherishThreshold	element	int	Port packet count threshold for the C/U class

7.2.2. Enhanced Delay-Loss

Enhanced Delay Loss scheduling based on probabilistic cherish/urgency distinction.

7.2.2.1. Policy set

- */policies/DIF/RMT/Monitor/eDLMonitor*
- */policies/DIF/RMT/MaxQueue/DumbMaxQ*
- */policies/DIF/RMT/Scheduler/DumbSch*

7.2.2.2. Configuration

Module	Variable	Description	Default value
eDLMonitor	xml cuData (array CUItem)	Cherish/Urgency classes definition	empty xml
eDLMonitor	xml urgData (array urgency)	Cherish/Urgency priority probability of skip	empty xml

CUItem - Cherish/Urgency Class definition

Parameter	Type	DataType	Description
id	attribute	string	Name of the C/U class
queue	element	string	Queue of the C/U class
urgency	element	int	Priority of the C/U class
cherishThreshold	element	int	Port packet count min-drop threshold for the C/U class

Parameter	Type	DataType	Description
cherishAbsThreshold	element	int	Port packet count absolute threshold for the C/U class
cherishDropProbability	element	double	Probability of drop between min-drop and absolute threshold for the C/U class

urgency - Cherish/Urgency priority probability of skip

Parameter	Type	DataType	Description
val	attribute	int	Priority
prob	attribute	double	Probability of skip

7.3. Routing

7.3.1. Distance Vector (legacy)

7.3.2. Link-state (legacy)

7.3.3. TSimple Link-state

Routing policy for QoS based routing domains. It allows to configure QoS named routing domains running a simple Link-State algorithm.

Extends "**IntTSimpleRouting**".

7.3.3.1. Policy set

- */policies/DIF/Routing/TSimpleLS*

7.3.3.2. Configuration

Module	Variable	Description	Default value
TSimpleLS	string myAddr	Node address in the DIF, for sending updates	""

Module	Variable	Description	Default value
TSimpleLS	bool printAtEnd	Print routing information at finish?	false

7.3.3.3. Interaction

Parameter Type <T> correspond to metric Type, defined by template. Currently TSimpleLS module sets T as unsigned short.

- void **insertFlow**(Address addr, string dst, string qos, T metric) Inserts or replaces a connection with QoS "qos" to a neighbour node. Neighbour defined with address in DIF "addr" and name "dst" within the routing domain. Metric of the connection "metric".
- void **removeFlow**(Address addr, string dst, string qos) Removes a connection for QoS "qos" to a neighbour node. Neighbour defined with address in DIF "addr" and name "dst" within the routing domain.
- map<string, map<string, nhLMetric<T> > > **getChanges**() Get changed next-hop entries for all domains after last query. Returned value in the form domain → dst Name → struct(T metric, set<string> next-hop)
- map<string, map<string, nhLMetric<T> > > **getAll**() Get all next-hop entries. Returned value in the form domain → dst Name → struct(T metric, set<string> next-hop)
- void **setInfinite**(T inf) Set the infinite metric to "inf".

7.3.4. TSimple Distance-vector

Routing policy for QoS based routing domains. It allows to configure QoS named routing domains running a simple Distance-Vector algorithm.

Extends "**IntTSimpleRouting**".

7.3.4.1. Policy set

- */policies/DIF/Routing/TSimpleDV*

7.3.4.2. Configuration

Module	Variable	Description	Default value
TSimpleDV	string myAddr	Node address in the DIF, for sending updates	""

Module	Variable	Description	Default value
TSimpleDV	bool printAtEnd	Print routing information at finish?	false

7.3.4.3. Interaction

Parameter Type <T> correspond to metric Type, defined by template. Currently TSimpleLS module sets T as unsigned short.

- void **insertFlow**(Address addr, string dst, string qos, T metric) Inserts or replaces a connection with QoS "qos" to a neighbour node. Neighbour defined with address in DIF "addr" and name "dst" within the routing domain. Metric of the connection "metric".
- void **removeFlow**(Address addr, string dst, string qos) Removes a connection for QoS "qos" to a neighbour node. Neighbour defined with address in DIF "addr" and name "dst" within the routing domain.
- map<string, map<string, nhLMetric<T> > > **getChanges**() Get changed next-hop entries for all domains after last query. Returned value in the form domain → dst Name → struct(T metric, set<string> next-hop)
- map<string, map<string, nhLMetric<T> > > **getAll**() Get all next-hop entries. Returned value in the form domain → dst Name → struct(T metric, set<string> next-hop)
- void **setInfinite**(T inf) Set the infinite metric to "inf".

7.3.5. Routing domain

Routing policy for configurable domains. It allows to configure named routing domains, for distinct QoS, sub-DIFs, etc., as well as decide which algorithm use within the domain (currently simple Link-State or Distance-Vector), node name and synonyms within the domain and neighbours in the domain.

7.3.5.1. Policy set

- */policies/DIF/Routing/TDomainRouting*

7.3.5.2. Configuration

Module	Variable	Description	Default value
TDomainRouting	string myAddr	Node address in the DIF, for sending updates	""

Module	Variable	Description	Default value
TDomainRouting	bool printAtEnd	Print routing information at finish?	false

7.3.5.3. Interaction

Parameter Type <T> correspond to metric Type, defined by template. Currently TDomainRouting module sets T as unsigned short.

- void **addDomain**(string domId, string addr, T infinite, ModuleAlgs alg) Define a new routing domain with name "domId". Self address "addr", infinite metric set at "infinite", and using routing algorithm "alg".
- void **removeDomain**(string domId) Removes the routing domain with name "domId".
- void **insertFlow**(Address addr, string dst, string domId, T metric) Inserts or replaces a connection to a neighbour node within domain "domId". Neighbour defined with address in DIF "addr" and name "dst" within the domain. Metric of the connection "metric".
- void **removeFlow**(Address addr, string dst, string domId) Removes a connection to a neighbour node within domain "domId". Neighbour defined with address in DIF "addr" and name "dst" within the domain.
- void **addAddr**(string domId, string syn) Add the synonim "syn" for the node in routing domain "domId".
- void **removeAddr**(string domId, string syn) Remove the synonim "syn" from the node in routing domain "domId".
- map<string, map<string, nhLMetric<T> > > **getChanges**() Get changed next-hop entries for all domains after last query. Returned value in the form domain $\rightarrow$ dst Name $\rightarrow$ struct(T metric, set<string> next-hop)
- map<string, map<string, nhLMetric<T> > > **getAll**() Get all next-hop entries. Returned value in the form domain $\rightarrow$ dst Name $\rightarrow$ struct(T metric, set<string> next-hop)

7.4. Forwarding

7.4.1. MiniTable

Simple forwarding policy based on a forwarding table storing a mapping dst addr $\rightarrow$ RMTPort*.

Extends "**IntMiniForwarding**".

7.4.1.1. Policy set

- */policies/DIF/RMT/PDUForwarding/MiniTable*

7.4.1.2. Configuration

Module	Variable	Description	Default value
MiniTable	bool printAtEnd	Print forwarding table at finish?	false

7.4.1.3. Interaction

- void **insert**(string addr, RMTPort * port) Inserts the entry "addr" → "port"
- void **insert**(Address addr, RMTPort * port) Synonym for insert(addr.getIpcAddress().getName(), port).
- void **remove**(string addr) Remove entry "addr".
- void **remove**(Address addr) Synonym for remove(addr.getIpcAddress().getName()).
- void **clean**() Clears forwarding table.

7.4.2. MultiMiniTable

Simple forwarding policy based on a forwarding table storing a mapping dst addr → vector<RMTPort*>. On lookup, it returns a random RMTPort* if more than one is available, resulting in a first/easy approach to load balancing.

Extends "**IntMMForwarding**".

7.4.2.1. Policy set

- */policies/DIF/RMT/PDUForwarding/MultiMiniTable*

7.4.2.2. Configuration

Module	Variable	Description	Default value
MiniTable	bool printAtEnd	Print forwarding table at finish?	false

7.4.2.3. Interaction

- void **addReplace**(string addr, vector<RMTPort *> ports) Sets entry "addr" as "ports". If "ports" is empty, removes entry "addr".

7.5. PDU Forwarding Table Generator

7.5.1. HopsSingle1Entry

PDU Forwarding Generator policy for hop based routing without distinction on QoS and only one dst Port per dst addr.

7.5.1.1. Policy set

- */policies/DIF/RA/PDUFG/HopsSingle1Entry*

7.5.1.2. Requires

- Forwarding policy implements IntMiniForwarding
- Routing policy IntTSimpleRouting<unsigned short>

7.5.1.3. Configuration

Module	Variable	Description	Default value
HopsSingle1Entry	unsigned short infinite	Infinite value for routing	32

7.5.2. HopsSingleMEntries

PDU Forwarding Generator policy for hop based routing without distinction on QoS and multiple ports per dst addr.

7.5.2.1. Policy set

- */policies/DIF/RA/PDUFG/HopsSingleMEntries*

7.5.2.2. Requires

- Forwarding policy implements IntMMForwarding
- Routing policy IntTSimpleRouting<unsigned short>

7.5.2.3. Configuration

Module	Variable	Description	Default value
HopsSingleMEntries	unsigned short infinite	Infinite value for routing	32

7.5.3. LatencySingle1Entry

PDU Forwarding Generator policy for latency based routing without distinction on QoS and only one dst Port per dst addr.

7.5.3.1. Policy set

- */policies/DIF/RA/PDUFG/LatencySingle1Entry*

7.5.3.2. Requires

- Forwarding policy implements IntMiniForwarding
- Routing policy IntTSimpleRouting<unsigned short>

7.5.3.3. Configuration

Module	Variable	Description	Default value
LatencySingle1Entry	unsigned short infinite	Infinite value for routing	1000
LatencySingle1Entry	unsigned short redLinkCost	Link Cost = QoS Latency / redLinkCost	1
LatencySingle1Entry	unsigned short maxLinkCost	Maximum Link Cost	100
LatencySingle1Entry	unsigned short minLinkCost	Minimum link cost	1

7.5.4. LatencySingleMEntries

PDU Forwarding Generator policy for latency based routing without distinction on QoS and multiple ports per dst addr.

7.5.4.1. Policy set

- */policies/DIF/RA/PDUFG/LatencySingleMEntries*

7.5.4.2. Requires

- Forwarding policy implements IntMiniForwarding

- Routing policy IntTSimpleRouting<unsigned short>

7.5.4.3. Configuration

Module	Variable	Description	Default value
LatencySingleMEntries	unsigned short infinite	Infinite value for routing	1000
LatencySingleMEntries	unsigned short redLinkCost	Link Cost = QoS Latency / redLinkCost	1
LatencySingleMEntries	unsigned short maxLinkCost	Maximum Link Cost	100
LatencySingleMEntries	unsigned short minLinkCost	Minimum link cost	1

8. Demonstration scenarios

This chapter outlines available examples of networks using RINA as native network stack. General instructions, how to setup and run scenarios, are provided to reader. Furthermore, detail description of notable scenarios try to reveal advantages of adopting RINA for certain Internet use-cases.

Source codes of demonstrations are located in */examples/* folder and each one includes following files, which may be used as templates when creating other RINASim scenarios:

- *<name>.ned* – OMNeT++ simulation network graph description which contains nodes and interconnections definitions;
- *omnetpp.ini* – scheduled simulation setup with models configuration (e.g., nodes addresses, ANI for AEs, pointers to XML configurations) applied during network initialization;
- *config.xml* – additional more structured and complex models configuration (e.g., DA's mappings, RA's QoS-cubes sets, preallocation and preenrollment settings) in the form of XML data is loaded to the simulation using this file;
- **.anf* – statistic collection setup file(s);
- *./results/* – results of various simulation runs containing gathered scalar and vector data.

Folder */playground/* contains various scenarios for testing purposes of their authors.

8.1. Running a Scenario

This assumes that OMNeT++ along with RINASim were correctly installed according to Chapter 3: Installation and Configuration.

8.1.1. From the IDE

1) Run the OMNeT++ IDE.

2) In the left pane, navigate to the folder with the desired example.

3) Right-click on omnetpp.ini, Run as#OMNeT++ simulation

4) Control the simulation via the Tkenv GUI, described in detail in the OMNeT++ User Guide [omnetpp-userguide].

- Note: For running the simulation via the console interface, change the User interface option in Run#Run configurations#<chosen example>.

8.1.2. From the Command Line

0) Prepare the console environment:

- on Windows: Execute the mingwenv.cmd batch file inside the OMNeT++ folder.
- On UN*X platforms: Open a console, navigate to the OMNeT++ folder and run `. ./setenv`.

1) Enter the root directory of RINASim.

2) Pick a folder with the desired example and run a simulation by one of the following ways:

- For CLI: `./simulate example_folder [-c configuration] [-x additional opp_run options]`

  # Note: if the -c argument is omitted, the simulation will default to configuration [General].

- For GUI: `./simulate example_folder -G [-c configuration] [-x additional opp_run options]`

8.2. Used Template

Each example except the first one has a fixed structure that contains the following items:

1. Brief motivation what could be observed in scenario
2. Picture of the scenario
3. Description of the events that may of interest for user
4. Initial simulation settings in *omnetpp.ini* file
5. Static XML configuration used to initialize RINA environment in *config.xml* file

8.3. Demo Network

Source files of this scenario are located in */examples/Demos/UseCase5*.

8.3.1. Motivation

This subchapter presents one of the many demonstration RINA simulations available in RINASim. The goals are: a) to give a reader overview of RINASim capabilities; and b) to familiarize the reader with RINA concepts on simple computer network example. The

motivation behind this particular simulation is to show ping-like application communication within the simple network consisting of all different node types.

8.3.2. Network Graph

Topology contains two host nodes (called *HostA* and *HostB*), two border routers (called *BorderRouterA* and *BorderRouterB*) and one interior router (called *InteriorRouter*) interconnected together as depicted in Figure below. Links between nodes are configured with one millisecond fixed transmission delay, which means that sending a packet from *HostA* to *HostB* takes four milliseconds.

There are totally six DIFs of three different ranks. Please notice addressing scheme where the same node may use the same address on different DIF as long as they are unambiguous within the layer's scope. RINA address length and syntax is policy-dependent. The demonstration uses flat address space with simple string addresses.

- Top most *TopLayer* DIF common to *HostA* (with address hA), *BorderRouterA* (address rA and self-enrolled), *BorderRouterB* (address rB) and *HostB* (hB);
- 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).

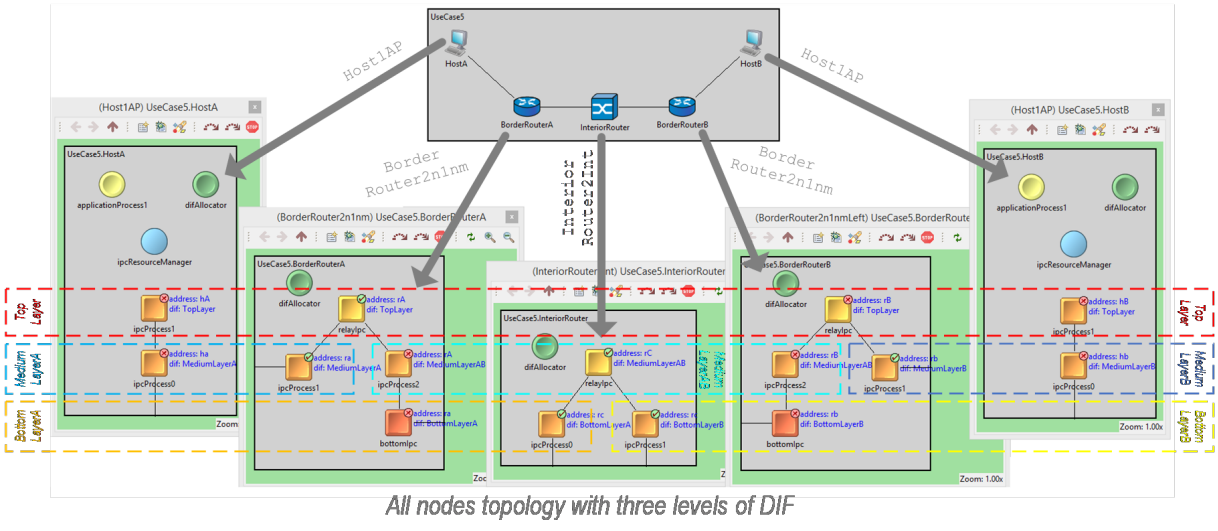

Figure 57. Demo network graph

8.3.3. Description

Multiple noticeable events happen during demonstration:

1. 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 IPC 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 rank DIFs.
2. The IPC comprises of flow allocation, data transfer, and optional flow deallocation. *HostA* and *HostB* are configured for IPC using ping-like application (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`.

By default, every RA contains implicit QoSCube (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 appropriate (N)-flow. Apart from this default QoS-cube, each RA loads 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” (same QoS parameters differing only in data transfer reliability). Please see figure below for visualization of loaded QoS-cube.

DA implementation currently allows only static change of its settings (namely different kinds of mappings). Hence, necessary configuration step is to initialize 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). Figure below shows shared directory information by all DA instances within the demonstration.

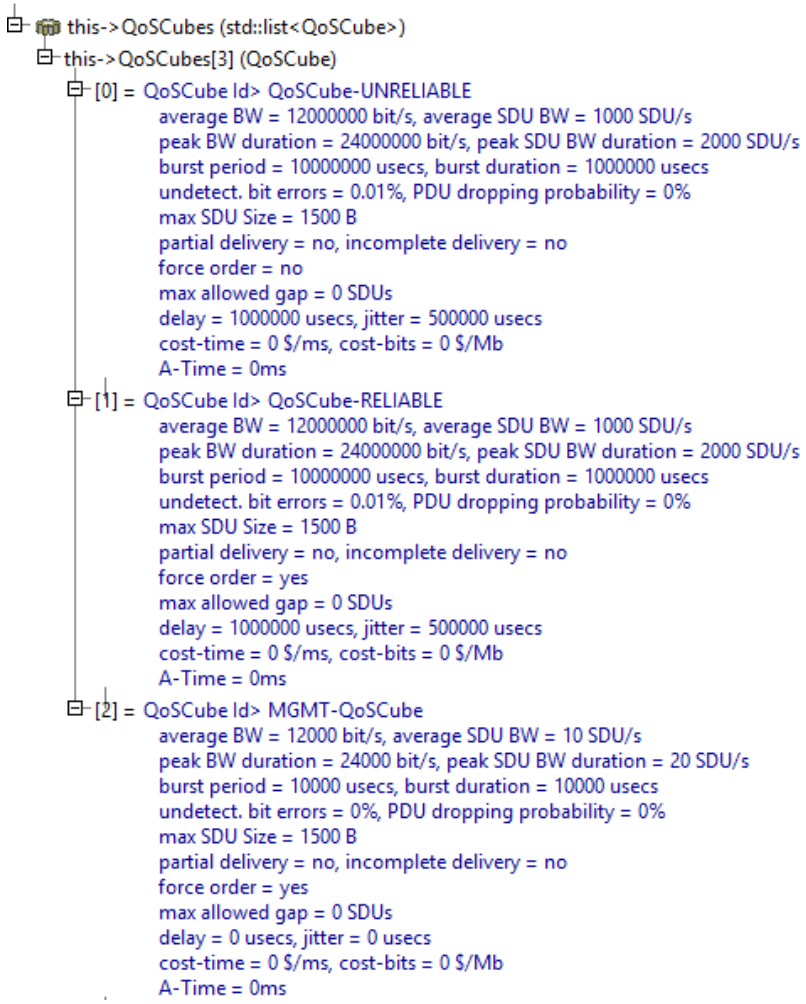

Figure 58. Visualization RA's available QoS-cubes

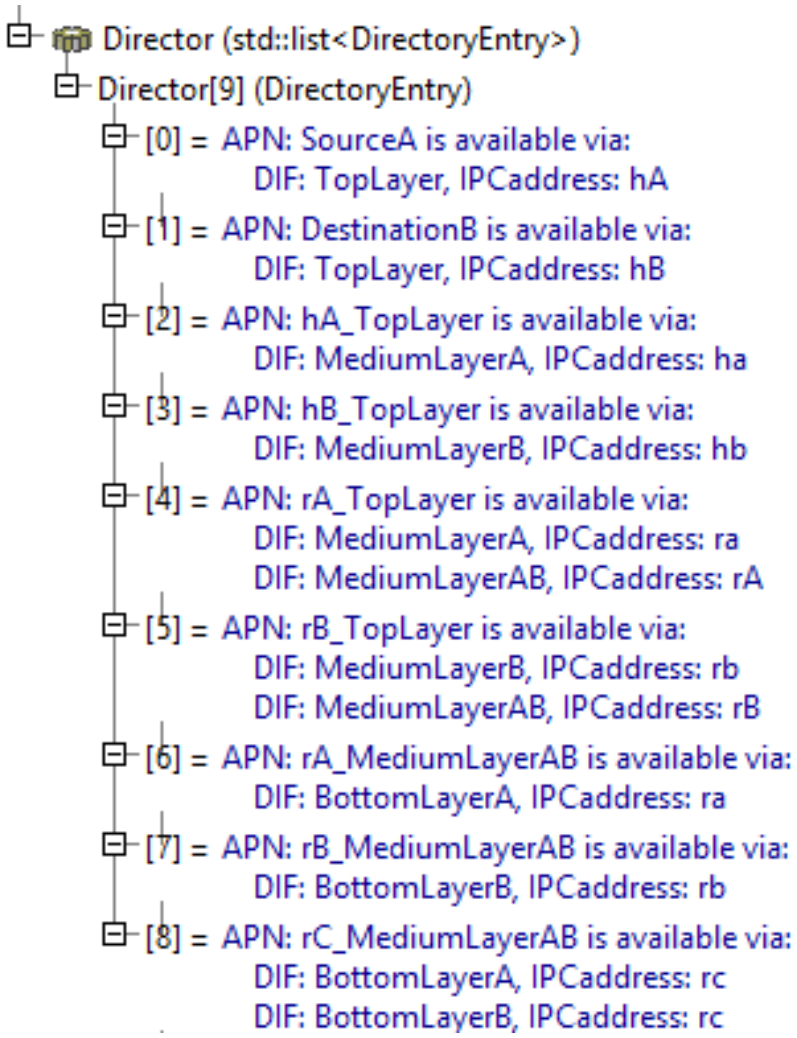

Figure 59. Visualization of Directory mappings

Simulation description is divided into two subsections. All events connected with enrollment procedures are described in “Enrollment Phase” subsection and events related to data transfer between *HostA* and *HostB* are in “Data Transfer Phase” subsection. The most important parts are descriptions of the trivial enrollment use-case (steps marked with #**), trivial flow allocation use-case (steps marked with #**), trivial recursion call (steps marked with #*). They outline steps, which repeat upon similar use-cases employing recursive calls.

8.3.3.1. Enrollment Phase

Whole enrollment phase is divided into four events. The first event is enrollment of *BorderRouterA* into *BottomLayerA* at `t=1s` with the help of *InteriorRouter* as enroller:

#1) *ipcProcess2*’s Enrollment module of *BorderRouterA* is scheduled to join the DIF *BottomLayerA* just a second after the simulation started. Enrollment asks FA to provide management (N-1)-flow (with destination address rc of *InteriorRouter*) to carry CACEP messages. Because *bottomIpc* is 0-level DIF (i.e., it is directly connected to the medium), then

RA returns automatically successful binding of the (N-1)-flow – recursion cannot continue below 0-level DIFs;

#2) *ipcProcess0*'s Enrollment sends M_CONNECT (with ra as source and rc as the destination address) via RIBd to *InteriorRouter*. *ipcProcess0*'s Enrollment module leverages IPCP with address rc within *BottomLayerA* (which is *bottomIpc* of *InteriorRouter*) when joining this DIF. Because management (N-1)-flow is inherently present, management messages can be sent immediately. M_CONNECT opens application connection for management messages between *BorderRouterA*'s *bottomIpc* and *InteriorRouter*'s *ipcProcess0*;

#3) *bottomIpc*'s Enrollment replies with positive M_CONNECT_R. With this message (sent from rc to ra), *bottomIpc* of *InteriorRouter* accepts application connection;

#4) *ipcProcess0*'s Enrollment begins enrollment procedure by sending M_START.

#5) *InteriorRouter* responds with M_START_R. Both of these messages contains EnrollmentObj as abstract data structure holding important parameters such as current address, address expiration time and APN. EnrollmentObj allows to assign dynamically address to newcoming DIF member. Nevertheless, this scenario works only with statically preconfigured addresses;

#6) Optionally, either *InteriorRouter* may send some M_CREATE messages to populate *BorderRouterA* RIB with information about neighboring IPCPs and their addresses. Alternatively, *BorderRouterA* may ask for this information using M_READ messages. Alternatively, alternatively, both can exchange some authentication objects proving the identity of communicating parties.

#7) However, let us consider the simplest case, where *bottomIpc*'s Enrollment sends M_STOP immediately after M_START_R. *InteriorRouter* ends enrollment procedure because it has all the necessary information from a joining member;

#8) *ipcProcess0*'s Enrollment replies with M_STOP_R. *BorderRouterA* finalizes enrollment by sending this message as Acknowledgement. The previous description outlines the most straightforward enrollment procedure that happens between joining member and enroller. The contents of EnrollmentStateTable (as abstract data structures holding information for IPCP's DIF membership) illustrating above-mentioned event is available in Addendum 8.5.3. Subsequent descriptions mention only notable changes because enrollment steps #1-#8 (CACEP message exchange) are present in all of them.

The second event is joining of *BorderRouterA* into *MediumLayerAB* at `t=1.5s` once again with the help of *InteriorRouter* as enroller:

#1) *BorderRouterA*'s *ipcProcess2* is scheduled with enrollment procedure to join *MediumLayerAB* leveraging *InteriorRouter*. Both IPCPs needs communication channel through which they may exchange management messages. Hence, *BorderRouterA*'s FA of *ipcProcess2* receives request for management flow (from Enrollment module) and asks RA to allocate appropriate (N-1)-flow (with source ra and destination rc) for communication with *InteriorRouter*'s IPCP with address rC;

#2) *ipcProcess2*'s RA bothers *bottomIpc*'s FA with allocation request because destination name resolution returned *bottomIpc* IPCP as being in the same DIF as IPCP with address rc. *bottomIpc*'s FA creates EFCPI to handle this data transfer (from perspective of *bottomIpc* this communication is just another data flow);

#3) *bottomIpc*'s FA sends M_CREATE containing Flow object inside via RIBd (because *bottomIpc* is already enrolled to the DIF *BottomLayerA*). Flow object describes all properties including source's and destination's addresses, port-ids, CEP-ids, QoS demands and chosen QoSCube (in case of management messages it is always predefined QoSCube with id "MGMT-QoSCube");

#4) M_CREATE is delivered to ipcProcess0's RIBd and FA, where it initiates the procedure for processing of create request flow. On *InteriorRouter*, *ipcProcess0* IPCP represents (N-1)-DIF for flow and *relayIpc* IPCP represents (N)-DIF for connection. Hence, ipcProcess0's FA notifies *relayIpc* about possible flow allocation. relayIpc's RIBd delegates this call to RA and Resource Allocator decides whether it has enough resources to accept or not the new flow.

#5) ipcProcess0's FA replies with positive M_CREATE_R. relayIpc's RA responded positively to allocation call. Therefore, ipcProcess0's FA instantiates opposite EFCPI, which involves the assignment of local port-id/CEP-id and bindings of gates. Following this, ipcProcess0's FA asks ipcProcess0's RIBd to generate and dispatch M_CREATE_R with updated Flow object stating successful flow allocation;

#6) *bottomIpc*'s FA receives M_CREATE_R and notifies *ipcProcess2*'s RA about it. FA updates local Flow object. Flow is effectively in place as a channel for communication between *BorderRouterA*'s *ipcProcess2* and *InteriorRouter*'s *relayIpc*. Hence, RA is alerted about (N-1)-flow being ready and handles control back to Enrollment module;

#7) Subsequently, steps #1-#8 repeats, where IPCP with address rA (*BorderRouterA*) is enrolled into *MediumLayerAB* by IPCP with address rC (*InteriorRouter*).

Create request/response flow calls are always accompanied by aforementioned steps #3-#7 and exchange of M_CREATE and M_CREATE_R messages. State information for each flow are stored in flowAllocator's submodule called nFlowTable. Illustration of relevant BottomLayerA's state tables is depicted in Figure below.

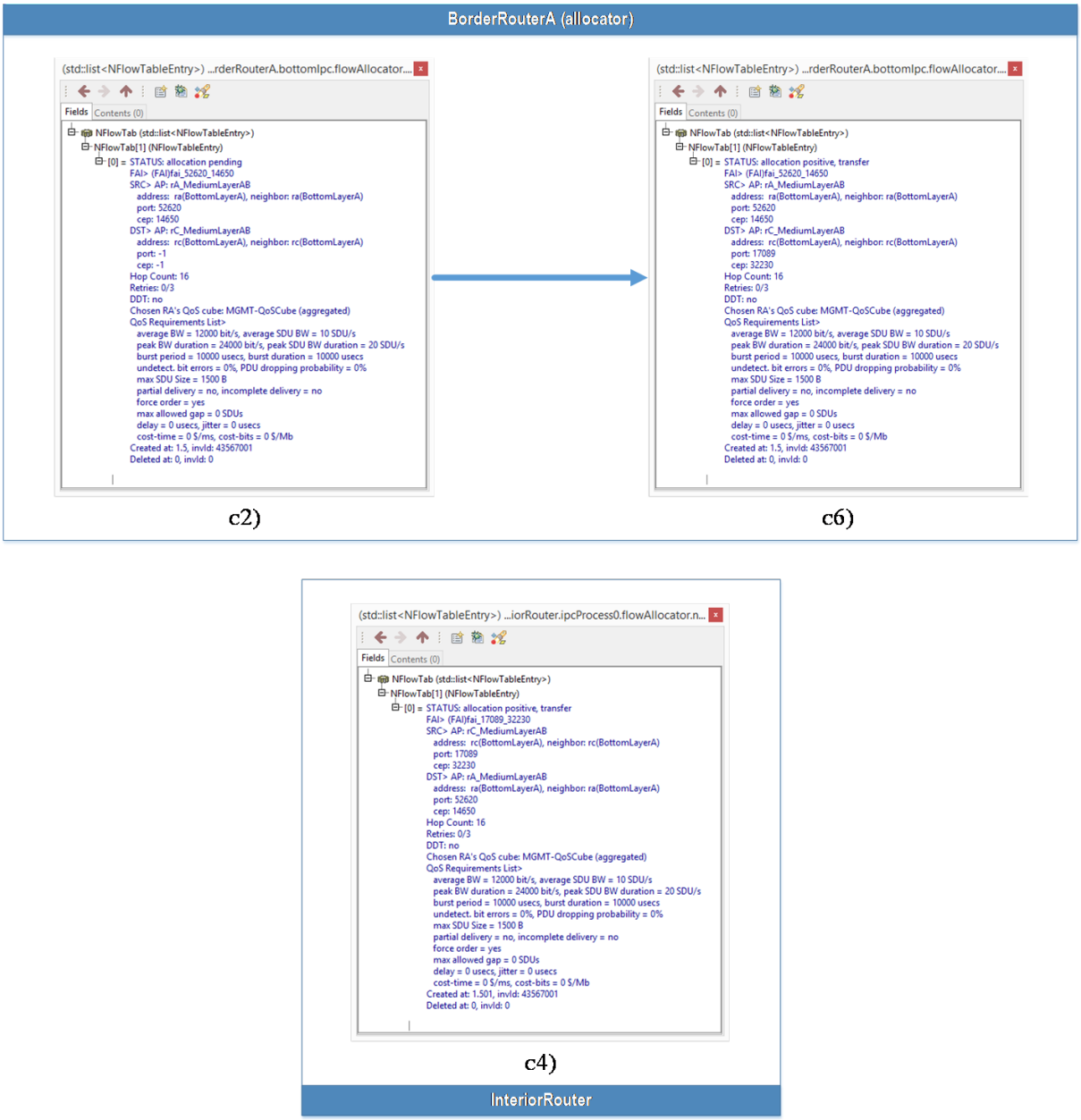

Figure 60. Content of BottomLayerA's NFlowTables of *BorderRouterA* and *InteriorRouter*

The third event is an enrollment of *BorderRouterB* into *TopLayer* at `t=2s`. Enrollment is scheduled on the top ranked IPCP (which is *relayIpc*) using *BorderRouterA* as enroller. Nevertheless, neither *BorderRouterB*'s *ipcProcess2*, nor *BorderRouterB*'s *bottomIpc* is enrolled to its DIF. Hence, *MediumLayerAB* enrollment must occur before *TopLayer* enrollment, and *BottomLayerB* enrollment must precede *MediumLayerAB* enrollment:

#1) relayIpc's Enrollment asks FA for management (N-1)-flow in order to send CACEP messages from rB to rA within *TopLayer*. Because it does not exist, RA delegates flow allocation to *ipcProcess2*;

#2) *ipcProcess2*'s FA receives a call. FA checks whether there is management (N-1)-flow for create request flow messages between rB (*BorderRouterB*'s *ipcProcess2*) and rA (*BorderRouterA*'s *ipcProcess2*) within *MediumLayerAB*. There is none flow and more over *BorderRouterB*'s *ipcProcess2* is not even enrolled into *MediumLayerAB*. Hence, *ipcProcess2*'s FA notifies RA that it need underlying management (N-1)-flow (from perspective of *relayIpc* it is (N-2)-flow) for enrollment procedure;

#3) *bottomIpc*'s FA receives a call. Because *bottomIpc* is in 0-level DIF, then RA returns automatically successful binding of the management (N-1)-flow. Enrollment procedure occurs between *BorderRouterB*'s *bottomIpc* and *InteriorRouter*'s *ipcProcess1*, which both are in *BottomLayerB* DIF. Basically, IPCP with address rb successfully enrolls into *BottomLayerB* using IPCP with address rc going through steps #1-#8;

#4) *bottomIpc*'s FA is notified about successful enrollment into *BottomLayerB* and continues with flow allocation initiated during step #3. Hence, *BorderRouterB*'s *bottomIpc* and *InteriorRouter*'s *ipcProcess1* RIBds and FAs exchange messages as in steps #3-#7. Eventually, management flow between rb and rc for *MediumLayerAB* communication is ready, and *BorderRouterB*'s *ipcProcess2* is alerted about this;

#5) *ipcProcess2*'s RA is notified about successful management flow allocation. Hence, enrollment procedure initiated in step #2 may continue. IPCP with address rB (*ipcProcess2* of *BorderRouterB*) successfully enrolls into *MediumLayerAB* using IPCP with address rC (*relayIpc* of *InteriorRouter*) going through steps #1-#8;

#6) *ipcProcess2*'s FA is notified about successful enrollment into *MediumLayerAB* and continues with flow allocation initiated during step #2. Hence, *BorderRouterB*'s *ipcProcess2* and *BorderRouterA*'s *ipcProcess2* exchange create request/response flow as in steps #3-#7. Notable difference comparing to flow allocation in step #4 is that messages pass through *InteriorRouter* (namely its *relayIpc*) as an interim device. Management flow between *InteriorRouter*'s *relayIpc* and *BorderRouterA*'s *ipcProcess2* is already present as the result of the second event of "Enrollment Phase". Eventually, management flow between rC and rA for *TopLayer* communication is in place, and *BorderRouterB*'s *relayIpc* is informed;

#7) relayIpc's RA is notified about successful management flow allocation. Hence, enrollment procedure initiated in step #1 may continue. All underlying connections are ready, and data path for management messages exists between *BorderRouterB* and *BorderRouterA* on relevant DIFs. IPCP with address rB (*relayIpc* of *BorderRouterB*) successfully enrolls into *TopLayer* using IPCP with address rA (*relayIpc* of *BorderRouterA*) going through steps #1-#8.

The fourth and the last event is an enrollment of *HostB* into *TopLayer* at `t=5s`. Enrollment is scheduled on the top ranked IPCP (which is *ipcProcess1*) using *BorderRouterB* as enroller.

Nevertheless, *BorderRouterB*'s *ipcProcess0* is also not enrolled into its DIF (*MediumLayerB*). Hence, *MediumLayerB* enrollment must occur before *TopLayer* enrollment. Situation is similar due to the recursions as in previous use-cases. Hence, we will omit unnecessary details when describing this event:

#1) *HostB*'s *ipcProcess1* checks existence of management (N-1) flow between *HostB*'s *ipcProcess0* and *BorderRouterB*'s *ipcProcess1*. There is none flow. Thus one must be allocated before enrollment procedure on *TopLayer*;

#2) Flow allocation call descend to *HostB*'s *ipcProcess0*. Over there is also as the first thing checked whether management (N-1) flow exists. Because *ipcProcess0* is in 0-level DIF, binding of (N-1)-flow is automatically successful;

#3) *HostB*'s *ipcProcess0* (with address hb) enrolls into *MediumLayerB* DIF using *BorderRouterB*'s *ipcProcess1* (with address rb) as enroller going through steps #1-#8;

#4) After *HostB* is successfully enrolled into *MediumLayerB*, management flow allocation from step #2 continues. The flow between *HostB*'s *ipcProcess0* and *BorderRouterB*'s *ipcProcess1* is created employing steps #3-#7. This flow is going to carry as data CACEP signalization messages between *HostB*'s *ipcProcess1* and *BorderRouterB*'s *relayIpc*;

#5) *HostB*'s *ipcProcess1* is notified about management (N-1) flow presence and enrollment procedure initiated in #1 continues. *HostB*'s *ipcProcess0* (with address hB) is enrolled into *TopLayer* DIF leveraging *BorderRouterB*'s *relayIpc* (with address rB).

The final state after "Enrollment Phase" is that all nodes IPCPs are enrolled (or self-enrolled) into their DIFs except HostA's IPCPs. All flows created during "Enrollment Phase" carries only CACEP 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.

8.3.3.2. Data Transfer Phase

The main outcome of this scenario is a simulation of data transfer events between *HostA* and *HostB* employing ping-like application (`AEMyPing`). This application 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 employing this simple application.

"Data Transfer Phase" is divided into three notable events – flow allocation, data transfer, and flow deallocation. We will describe them in similar fashion as the previous phase. Data flow allocation starts at `t=10s` . HostA's applicationProcess1 (with APN SourceA, API-id 0, AEN MyPing, AE-id 0 as ANI parameters) requests flow for communication with

HostB's applicationProcess1 (with APN DestinationB, API-id 0, AEN MyPing, AE-id 0 as ANI parameters). Event goes through following set of steps:

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

#2) *HostA* can access *TopLayer* leveraging *ipcProcess1*. Hence, IRM delegates allocate request call to ipcProcess1's FA. As usually, FA instantiates EFCPI and verifies whether IPCP is enrolled into DIF before any attempt for sending create request flow (analogous to steps #1-#2). The situation is now similar to enrollment procedure of *HostB* because neither *ipcProcess1* nor *ipcProcess0* are enrolled into their DIFs. Therefore, *HostA* repeats same steps #1-5, which involve following actions performed due to the recursive calls in this order of finalization: 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 flow allocation. FA exchanges create request/respond flow with *HostB* (analogously to #3-#7). This includes the creation of (N-1)-flow between ha and ra in *MediumLayerA* and creation of (N)-flow between hA and hB in *TopLayer*. However, it gets more complex in *TopLayer* DIF because M_CREATE and M_CREATE_R messages must be relayed by 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 established flows serve as carriers for *HostA* and *HostB* data transfer. The next steps briefly describe this multi-action step;

#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 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, 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 flow and sends create respond flow that travels back to *BorderRouterA*. Because 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 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 flow between rb and hb in *MediumLayerB*. Subsequently, M_CREATE is forwarded to *HostB*;

#8) M_CREATE is received by *HostB*'s *ipcProcess1*. FA notifies applicationProcess1 about ongoing flow allocation. applicationProcess1 accepts flow for data transfer between APs. The decision is returned to ipcProcess1's FA. IRM is asked to create bindings between AP and IPCP. FA instantiates EFCPI, updates Flow object and replies back to 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. FA updates Flow object and notifies applicationProcess1 about successful flow allocation. Then IRM adds missing bindings and whole data path between *HostA* and *HostB* is ready. (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 probes employing own object inside M_READ message starting at `t=15s`. Upon reception of these messages, *HostB* replies with probe response, which is dedicated M_READ_R message. Data path and relevant flows are depicted in with different colors to get oriented in the following the description. Event consists of five repetitions of 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, message ascends to *relayIpc*, where 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 flow in *ipcProcess1* (member of TopLayerB) and through IRM to *HostB*'s applicationProcess1 as 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.

Depending on direction 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).

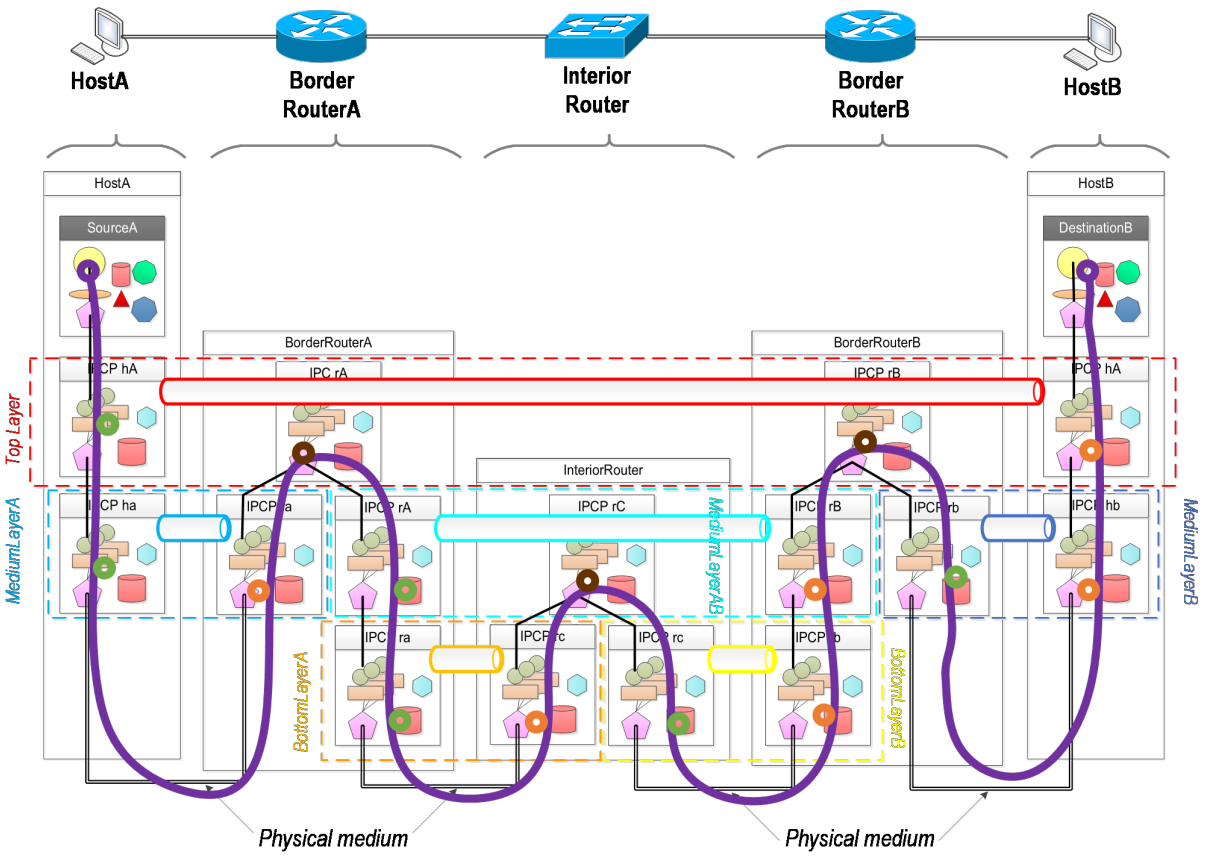

Figure 61. Data transfer phase illustration

After APs exchanged pings, HostA's AE closes the connection and sends deallocate submit to *HostB* at `t=20s`. Deallocation affects only flow present in *TopLayer*. Current RINASim implementation leaves 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 deallocate submit. IRM disconnects from its side port binding. Then, IRM delegates flow deallocation to ipcProcess1's FA;

#2) This FA generates a M_DELETE message with updated Flow object state inside and sends it towards *HostB* through flow in *TopLayer*. Message follows data path leveraging existing management flows created during enrollment phase;

#3) *HostB*'s *ipcProcess1* receives M_DELETE. FA updates its version of Flow object. FA delivers deallocation submit to *HostB*'s applicationProcess1, which tells IRM to remove bindings.

#4) ipcProcess1's FA on *HostB* then replies with M_DELETE_R acknowledging successful flow deallocation. This message is carried back to *HostA*;

#5) HostA's *ipcProcess1* receives M_DELETE_R. FA marks 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*. We can inspect flow parameters in these tables as illustrated in figure below. We can see that two EFCPIs handled endpoints of data transfer – EFCPI with CEP-id 18 430 in HostA's *ipcProcess1* and EFCPI with CEP-id 60 067 in *HostB*'s *ipcProcess1*. Bindings between AP and IPCP are ports identified with port-id 7 877 for *HostA* and port-id 57 495 for *HostB*. The only QoS demand by AEMyPing is the reliability of data transfer (expressed with QoS attribute "force order" set to true). Therefore, RA assigned QoSCube named "QoSCube-RELIABLE" to flows requested by this AE. Flow object between *HostA* and *HostB* in *TopLayer* was created at `t=10s/10.026s` and was deleted at t=20.008s/20.004s.

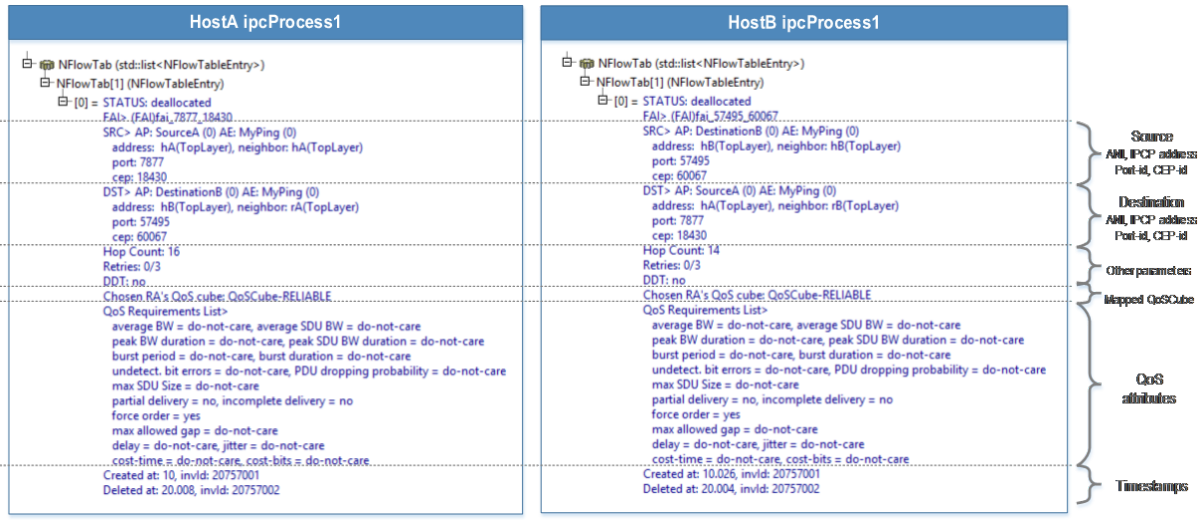

Figure 62. Content of *TopLayer ipcProcess1* NFlowTables for *HostA* and *HostB*

8.3.4. omnetpp.ini

```
[General]
network = UseCase5
check-signals = true
sim-time-limit = 5min
debug-on-errors = true
#Application setup
**.HostA.applicationProcess1.apName = "SourceA"
**.HostB.applicationProcess1.apName = "DestinationB"
**.iae.aeName = "MyPing"
**.applicationEntity.aeType = "AEMyPing"
```

```
#DIF Naming
**.Host*.ipcProcess1.difName         = "TopLayer"
**.BorderRouter*.relayIpc.difName    = "TopLayer"
**.HostA.ipcProcess0.difName         = "MediumLayerA"
**.BorderRouterA.ipcProcess1.difName = "MediumLayerA"
**.HostB.ipcProcess0.difName         = "MediumLayerB"
**.BorderRouterB.ipcProcess1.difName = "MediumLayerB"
**.BorderRouterA.ipcProcess2.difName = "MediumLayerAB"
**.InteriorRouter.relayIpc.difName   = "MediumLayerAB"
**.BorderRouterB.ipcProcess2.difName = "MediumLayerAB"
**.BorderRouterA.bottomIpc.difName   = "BottomLayerA"
**.InteriorRouter.ipcProcess0.difName= "BottomLayerA"
**.BorderRouterB.bottomIpc.difName   = "BottomLayerB"
**.InteriorRouter.ipcProcess1.difName= "BottomLayerB"

#Static IPC Addressing
**.HostA.ipcProcess1.ipcAddress  = "hA"
**.HostB.ipcProcess1.ipcAddress  = "hB"
**.BorderRouterA.relayIpc.ipcAddress  = "rA"
**.BorderRouterB.relayIpc.ipcAddress  = "rB"
**.HostA.ipcProcess0.ipcAddress  = "ha"
**.BorderRouterA.ipcProcess1.ipcAddress = "ra"
**.HostB.ipcProcess0.ipcAddress  = "hb"
**.BorderRouterB.ipcProcess1.ipcAddress = "rb"
**.BorderRouterA.ipcProcess2.ipcAddress = "rA"
**.InteriorRouter.relayIpc.ipcAddress = "rC"
**.BorderRouterB.ipcProcess2.ipcAddress = "rB"
**.BorderRouterA.bottomIpc.ipcAddress  = "ra"
**.InteriorRouter.ipcProcess0.ipcAddress= "rc"
**.BorderRouterB.bottomIpc.ipcAddress  = "rb"
**.InteriorRouter.ipcProcess1.ipcAddress= "rc"

#DIF Allocator settings
**.HostA.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='HostA']/DA")
**.HostB.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='HostB']/DA")
**.BorderRouterA.difAllocator.configData = xmldoc("config.xml",
 "Configuration/Router[@id='BorderRouterA']/DA")
**.BorderRouterB.difAllocator.configData = xmldoc("config.xml",
 "Configuration/Router[@id='BorderRouterB']/DA")
**.InteriorRouter.difAllocator.configData = xmldoc("config.xml",
 "Configuration/Router[@id='InteriorRouter']/DA")
**.HostB.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='HostA']/DA")
```

```
**.BorderRouterA.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='HostA']/DA")
**.BorderRouterB.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='HostA']/DA")
**.InteriorRouter.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='HostA']/DA")

#Enrollment settings
**.InteriorRouter.**.enrollment.isSelfEnrolled = true
**.BorderRouterA.relayIpc.**.enrollment.isSelfEnrolled = true
**.BorderRouterA.ipcProcess1.**.enrollment.isSelfEnrolled = true
**.BorderRouterB.ipcProcess1.**.enrollment.isSelfEnrolled = true
**.BorderRouterA.bottomIpc.enrollment.configData = xmldoc("config.xml",
 "Configuration/Router[@id='BorderRouterA']/Enrollment[@id='bottomIpc']")
**.BorderRouterA.ipcProcess2.enrollment.configData =
 xmldoc("config.xml", "Configuration/Router[@id='BorderRouterA']/
Enrollment[@id='ipcProcess2']")
**.BorderRouterB.relayIpc.enrollment.configData = xmldoc("config.xml",
 "Configuration/Router[@id='BorderRouterB']/Enrollment[@id='relayIpc']")
**.HostB.ipcProcess1.enrollment.configData = xmldoc("config.xml",
 "Configuration/Host[@id='HostB']/Enrollment")

#QoS Cube sets
**.ra.qoscubesData = xmldoc("config.xml", "Configuration/QoSCubesSet")

[Config Ping]
#PingApp setup
**.forceOrder = true
**.HostA.applicationProcess1.applicationEntity.iae.dstApName =
 "DestinationB"
**.HostA.applicationProcess1.applicationEntity.iae.dstAeName = "MyPing"
**.HostA.applicationProcess1.applicationEntity.iae.startAt = 10s
**.HostA.applicationProcess1.applicationEntity.iae.pingAt =  15s
**.HostA.applicationProcess1.applicationEntity.iae.rate = 5
**.HostA.applicationProcess1.applicationEntity.iae.stopAt = 20s
**.HostA.applicationProcess1.applicationEntity.iae.size = 1024B
```

8.3.5. config.xml

```
<?xml version="1.0"?>
<Configuration>

 <Host id="HostA">
  <DA>
   <Directory>
```

```
    <APN apn="SourceA">
     <DIF difName="TopLayer" ipcAddress="hA" />
    </APN>
    <APN apn="DestinationB">
     <DIF difName="TopLayer" ipcAddress="hB" />
    </APN>

    <APN apn="hA_TopLayer">
     <DIF difName="MediumLayerA" ipcAddress="ha" />
    </APN>
    <APN apn="hB_TopLayer">
     <DIF difName="MediumLayerB" ipcAddress="hb" />
    </APN>

    <APN apn="rA_TopLayer">
     <DIF difName="MediumLayerA" ipcAddress="ra" />
     <DIF difName="MediumLayerAB" ipcAddress="rA" />
    </APN>
    <APN apn="rB_TopLayer">
     <DIF difName="MediumLayerB" ipcAddress="rb" />
     <DIF difName="MediumLayerAB" ipcAddress="rB" />
    </APN>

    <APN apn="rA_MediumLayerAB">
     <DIF difName="BottomLayerA" ipcAddress="ra" />
    </APN>
    <APN apn="rB_MediumLayerAB">
     <DIF difName="BottomLayerB" ipcAddress="rb" />
    </APN>
    <APN apn="rC_MediumLayerAB">
     <DIF difName="BottomLayerA" ipcAddress="rc" />
     <DIF difName="BottomLayerB" ipcAddress="rc" />
    </APN>
   </Directory>
   <NeighborTable>
    <APN apn="hA_TopLayer">
     <Neighbor apn="rA_TopLayer" />
    </APN>
    <APN apn="hB_TopLayer">
     <Neighbor apn="rA_TopLayer" />
    </APN>
   </NeighborTable>
  </DA>
 </Host>

 <Host id="HostB">
```

```
 <DA>
  <NeighborTable>
   <APN apn="hA_TopLayer">
    <Neighbor apn="rB_TopLayer" />
   </APN>
   <APN apn="hB_TopLayer">
    <Neighbor apn="rB_TopLayer" />
   </APN>
  </NeighborTable>
 </DA>
 <Enrollment>
  <Preenrollment>
            <SimTime t="5">
                <Connect src="hB_TopLayer" dst="rB_TopLayer" />
            </SimTime>
  </Preenrollment>
 </Enrollment>
</Host>

<Router id="BorderRouterA">
 <DA>
  <NeighborTable>
   <APN apn="hB_TopLayer">
    <Neighbor apn="rB_TopLayer" />
   </APN>
   <APN apn="rB_MediumLayerAB">
    <Neighbor apn="rC_MediumLayerAB" />
   </APN>
  </NeighborTable>
 </DA>
 <Enrollment id='bottomIpc'>
  <Preenrollment>
            <SimTime t="1">
                <Connect src="ra_BottomLayerA" dst="rc_BottomLayerA" />
            </SimTime>
  </Preenrollment>
 </Enrollment>
 <Enrollment id='ipcProcess2'>
  <Preenrollment>
            <SimTime t="1.5">
                <Connect src="rA_MediumLayerAB" dst="rC_MediumLayerAB" />
            </SimTime>
  </Preenrollment>
 </Enrollment>
</Router>
```

```
<Router id="BorderRouterB">
 <DA>
  <NeighborTable>
   <APN apn="hA_TopLayer">
    <Neighbor apn="rA_TopLayer" />
   </APN>
   <APN apn="rA_MediumLayerAB">
    <Neighbor apn="rC_MediumLayerAB" />
   </APN>
  </NeighborTable>
 </DA>
 <Enrollment id='relayIpc'>
  <Preenrollment>
            <SimTime t="2">
                <Connect src="rB_TopLayer" dst="rA_TopLayer" />
            </SimTime>
  </Preenrollment>
 </Enrollment>
</Router>

<Router id="InteriorRouter">
 <DA>
  <NeighborTable>
   <APN apn="hA_TopLayer">
    <Neighbor apn="rB_TopLayer" />
   </APN>
  </NeighborTable>
 </DA>
</Router>


<QoSCubesSet>
 <QoSCube id="QoSCube-UNRELIABLE">
  <AverageBandwidth>12000000</AverageBandwidth>
  <AverageSDUBandwidth>1000</AverageSDUBandwidth>
  <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
  <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
  <BurstPeriod>10000000</BurstPeriod>
  <BurstDuration>1000000</BurstDuration>
  <UndetectedBitError>0.01</UndetectedBitError>
  <PDUDroppingProbability>0</PDUDroppingProbability>
  <MaxSDUSize>1500</MaxSDUSize>
  <PartialDelivery>0</PartialDelivery>
  <IncompleteDelivery>0</IncompleteDelivery>
  <ForceOrder>0</ForceOrder>
  <MaxAllowableGap>0</MaxAllowableGap>
```

```
    <Delay>1000000</Delay>
    <Jitter>500000</Jitter>
    <CostTime>0</CostTime>
    <CostBits>0</CostBits>
    <ATime>0</ATime>
   </QoSCube>
   <QoSCube id="QoSCube-RELIABLE">
    <AverageBandwidth>12000000</AverageBandwidth>
    <AverageSDUBandwidth>1000</AverageSDUBandwidth>
    <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
    <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
    <BurstPeriod>10000000</BurstPeriod>
    <BurstDuration>1000000</BurstDuration>
    <UndetectedBitError>0.01</UndetectedBitError>
    <PDUDroppingProbability>0</PDUDroppingProbability>
    <MaxSDUSize>1500</MaxSDUSize>
    <PartialDelivery>0</PartialDelivery>
    <IncompleteDelivery>0</IncompleteDelivery>
    <ForceOrder>1</ForceOrder>
    <MaxAllowableGap>0</MaxAllowableGap>
    <Delay>1000000</Delay>
    <Jitter>500000</Jitter>
    <CostTime>0</CostTime>
    <CostBits>0</CostBits>
    <ATime>0</ATime>
   </QoSCube>
  </QoSCubesSet>
 </Configuration>
```

8.4. Demonstration: Congestion

8.4.1. Motivation

The way RINA controls congestion is a generalization of how it is done in the Internet: if there is only one DIF doing congestion control in the network, it operates in an end-to-end fashion. If two or more congestion controlled DIFs are concatenated, the end-to-end control loop is broken into shorter loops. As another interesting capability, RINA allows DIFs to be stacked, and upper DIFs can have their own congestion control policies. If there are several flows from one sender to one receiver through several EFCP connections, packets of all of them are mapped to only one EFCP connection in the DIFs below; this means that at the lower DIFs, there is only one aggregated flow, and congestion control in these DIFs operates on aggregates. In a real RINA network where DIFs are stacked above each other, an N-DIF would carry an aggregate of flows from the (N+1)-DIF sitting above it. Edge router pairs would then only keep

the congestion state of active flow aggregates between them. Here, RINA-ACC automatically avoids the competition between multiple end-to-end flows that occurs in the Internet today.

The goal of this demonstration is to show how RINA's Aggregate Congestion Control (ACC) policies are used in a simple network topology. In particular, we show how flows can be aggregated and controlled using one congestion controller to reduce the negative effect of competing flows for a shared bandwidth on each other.

8.4.2. Description

To achieve the above goal, we simulated a scenario in which multiple flows were sharing the same bottleneck link in the network. The example is named SmallNetwork3 in the example folder of RINASim. The network topology and its RINA stack are shown in Figure 63 and Figure 64, respectively. host1x sends a large file to host2x, respectively. The link between $Router_1$ and $Router_2$ was the bottleneck link. There was one lower-layer DIF per each link, and one upper-layer DIF on top of them. The RMTs used *UpstreamNotifier*, and the set of *TxControlPolicyTCPTahoe*, *RTTEstimatorPolicyTCP*, and *SenderAckPolicyTCPTahoe* ACC policies was used as the congestion controller in IPCPs.

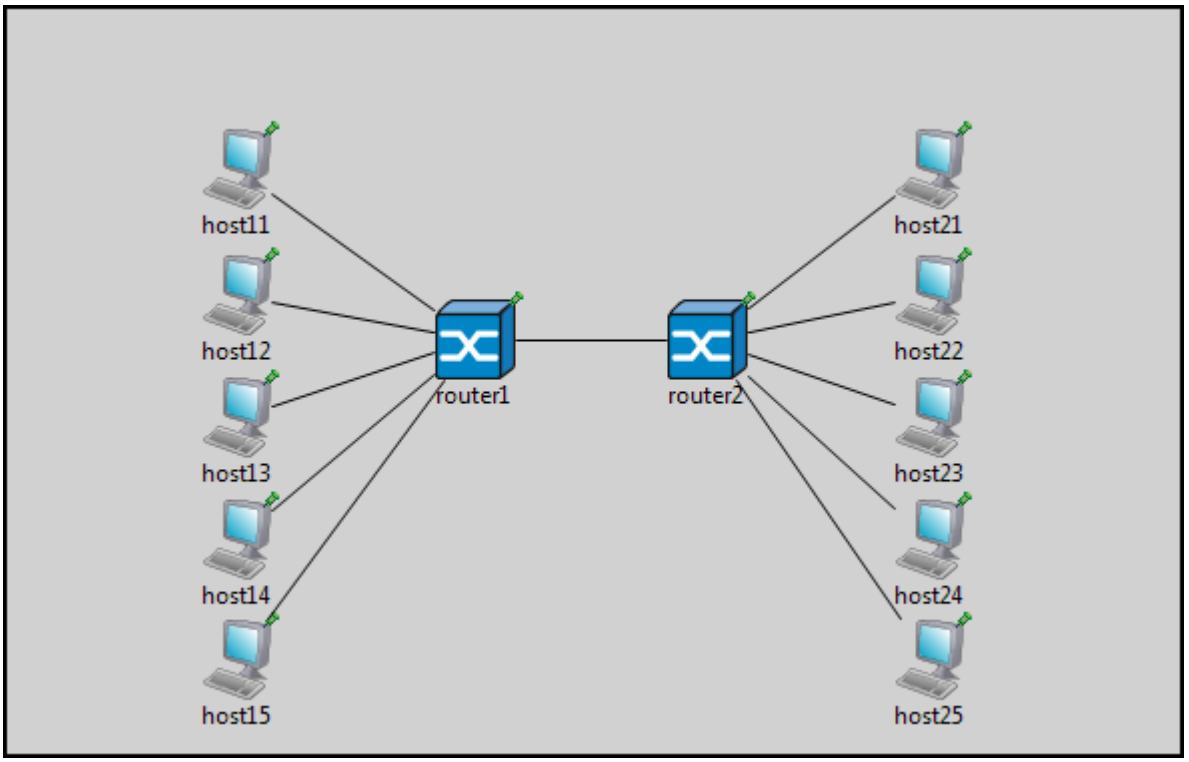

Figure 63. Network topology

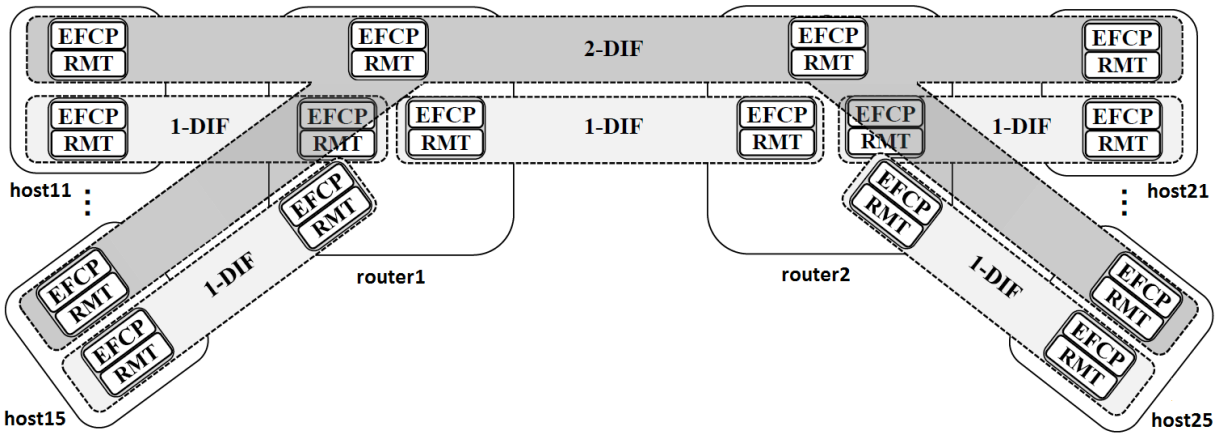

Figure 64. The corresponding RINA stack

8.4.3. Major events

After running the sample in OMNeT++, the following events happen which are worth mentioning. The time unit is second.

- At t = 2, all sender nodes, host1x, start transmission.
- At t = 3.88, the bottleneck link is fully utilized and the output buffer in $Router_1$, the RMT output queue of the lower DIF, builds up.
- At t = 3.97, the RMT output queue reaches its threshold, which in turn, sends a notification in the EFCP module in the same IPCP to slow down. Upon getting the slow down signal, the EFCP instance calls the slow down method in DTCP, which reduces the transmission window.
- At t = 3.98, the closed window queue of DTP builds up.
- At t = 4.05, the RMT output queue of the upper DIF in $Router_1$ is built up and exceeds its threshold. This initiates another pushback signal to the sender EFCP instance of the last packet on the queue. In this case, the EFCPI is in the top DIF of one of the sender nodes. The signal is converted to a pushback packet and sent towards the sender node.
- At t = 4.08, the corresponding sender node gets the pushback signal through ECNSlowDown policy and consequently, *TxControlPolicyTCPTahoe* reduces its send window. The above series events happen for all the other senders during the simulation until it finishes.
- At t = 62, the simulation ends; all sender nodes stop transmission, and statistics is collected. The results are in the Aggregation-0.vec, Aggregation-0.vci, and Aggregation-0.sca files. By creating the Aggregation.anf file, results are observable.

The following diagrams are generated automatically by the Aggregation.anf file. The congestion window size (in Bytes) of the senders and the EFCP instance in $Router_1$ is shown in Figure 65. The upper line in the diagram belongs to the window size of the EFCP instance in $Router_1$.

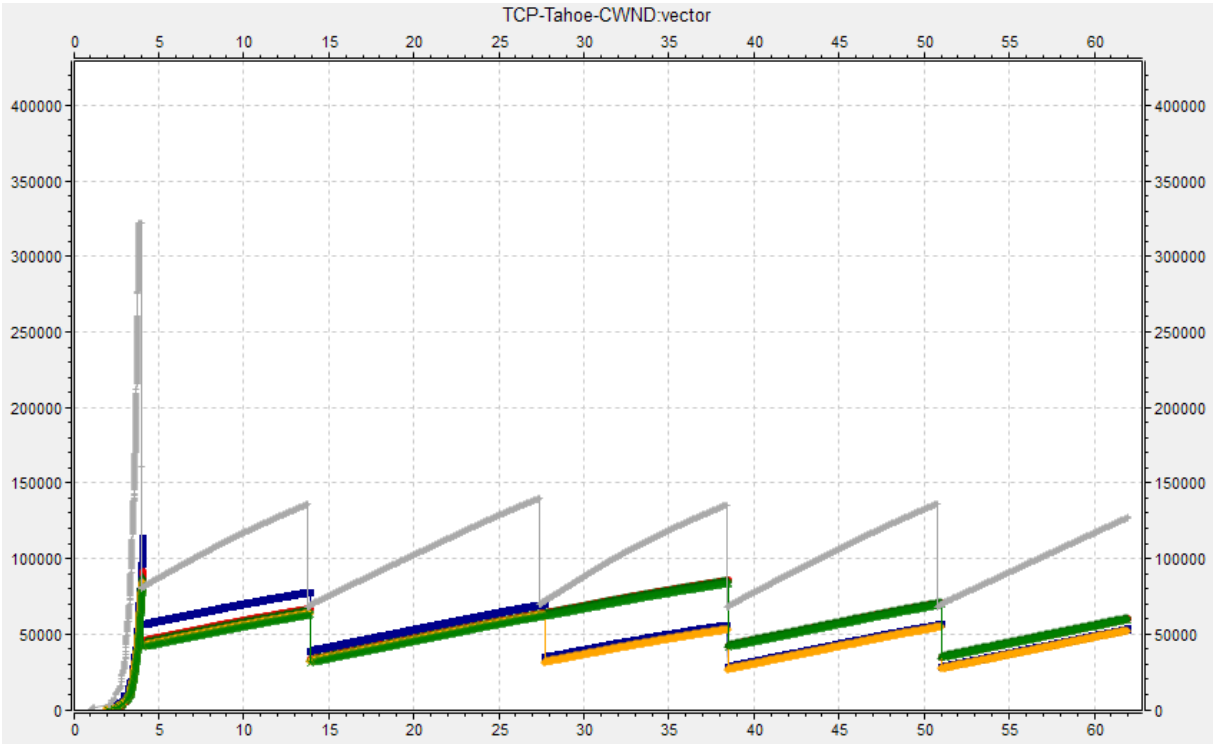

Figure 65. The congestion window size

The queue length of the RMT output queue in $Router_1$ in the lower and upper DIF is illustrated in Figure 66. The red curve belongs to the RMT queue in the upper DIF.

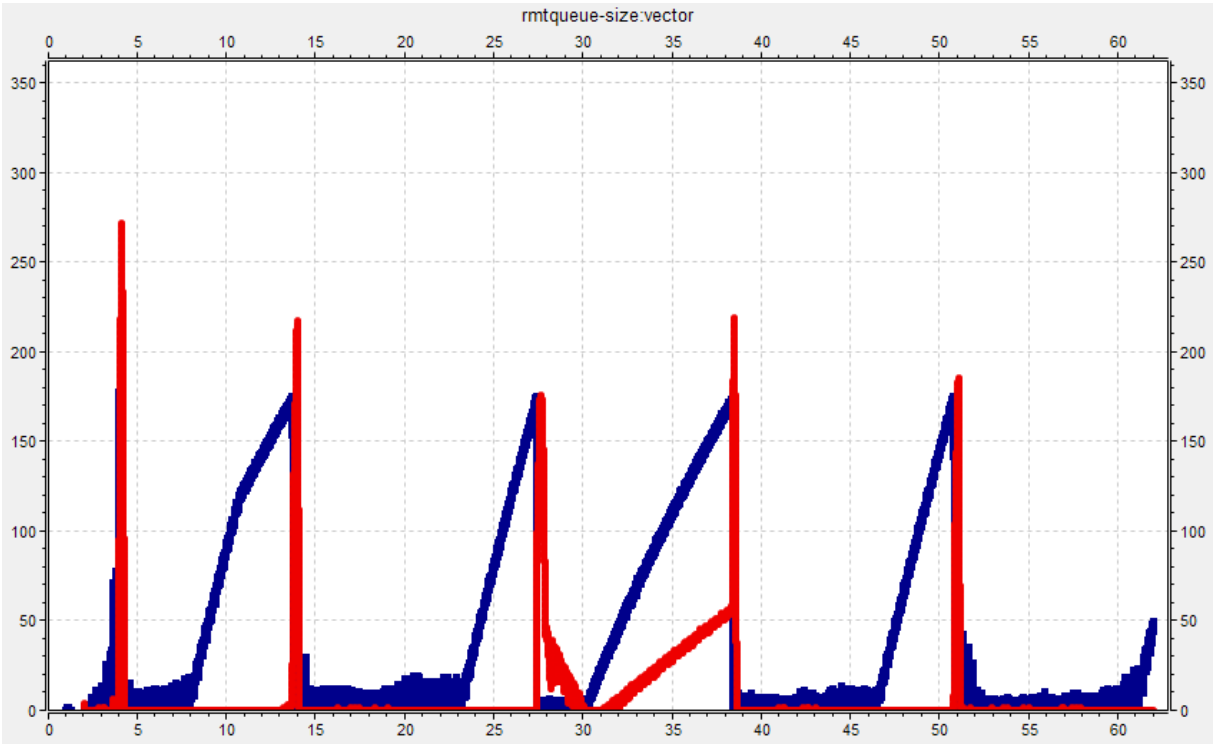

Figure 66. The RMT queue length

Taking a look at the Scalars tab in the Aggregation.anf file, the AE-PING-BYTES-RCVD:last values, in particular, reveals that the receivers got 1.2966376E7, 1.0637992E7, 1.3288512E7, 1.3411792E7, and 1.1893304E7 Bytes, respectively.

8.4.4. omnetpp.ini

```
[General]

sim-time-limit = 62s
seed-set = ${runnumber}
**.vector-recording = true

**.applicationEntity.aeType = "AEStream"

**.host11.applicationProcess1.apName = "App11"
**.host12.applicationProcess1.apName = "App12"
**.host13.applicationProcess1.apName = "App13"
**.host14.applicationProcess1.apName = "App14"
**.host15.applicationProcess1.apName = "App15"

**.host21.applicationProcess1.apName = "App21"
**.host22.applicationProcess1.apName = "App22"
**.host23.applicationProcess1.apName = "App23"
**.host24.applicationProcess1.apName = "App24"
**.host25.applicationProcess1.apName = "App25"

**.host11.applicationProcess1.applicationEntity.iae.aeName = "Stream11"
**.host12.applicationProcess1.applicationEntity.iae.aeName = "Stream12"
**.host13.applicationProcess1.applicationEntity.iae.aeName = "Stream13"
**.host14.applicationProcess1.applicationEntity.iae.aeName = "Stream14"
**.host15.applicationProcess1.applicationEntity.iae.aeName = "Stream15"

**.host21.applicationProcess1.applicationEntity.iae.aeName = "Stream21"
**.host22.applicationProcess1.applicationEntity.iae.aeName = "Stream22"
**.host23.applicationProcess1.applicationEntity.iae.aeName = "Stream23"
**.host24.applicationProcess1.applicationEntity.iae.aeName = "Stream24"
**.host25.applicationProcess1.applicationEntity.iae.aeName = "Stream25"

#Static addressing: lower IPC layer
**.host11.ipcProcess0.ipcAddress = "011"
**.host12.ipcProcess0.ipcAddress = "012"
**.host13.ipcProcess0.ipcAddress = "013"
**.host14.ipcProcess0.ipcAddress = "014"
**.host15.ipcProcess0.ipcAddress = "015"
```

```
**.host21.ipcProcess0.ipcAddress = "021"
**.host22.ipcProcess0.ipcAddress = "022"
**.host23.ipcProcess0.ipcAddress = "023"
**.host24.ipcProcess0.ipcAddress = "024"
**.host25.ipcProcess0.ipcAddress = "025"

**.router1.ipcProcess[0].ipcAddress = "031"
**.router1.ipcProcess[1].ipcAddress = "032"
**.router1.ipcProcess[2].ipcAddress = "033"
**.router1.ipcProcess[3].ipcAddress = "034"
**.router1.ipcProcess[4].ipcAddress = "035"
**.router1.ipcProcess[5].ipcAddress = "036"

**.router2.ipcProcess[0].ipcAddress = "046"
**.router2.ipcProcess[1].ipcAddress = "041"
**.router2.ipcProcess[2].ipcAddress = "042"
**.router2.ipcProcess[3].ipcAddress = "043"
**.router2.ipcProcess[4].ipcAddress = "044"
**.router2.ipcProcess[5].ipcAddress = "045"


**.host11.ipcProcess0.difName = "Layer011"
**.router1.ipcProcess[0].difName = "Layer011"

**.host12.ipcProcess0.difName = "Layer012"
**.router1.ipcProcess[1].difName = "Layer012"

**.host13.ipcProcess0.difName = "Layer013"
**.router1.ipcProcess[2].difName = "Layer013"

**.host14.ipcProcess0.difName = "Layer014"
**.router1.ipcProcess[3].difName = "Layer014"

**.host15.ipcProcess0.difName = "Layer015"
**.router1.ipcProcess[4].difName = "Layer015"

**.router1.ipcProcess[5].difName = "Layer034"
**.router2.ipcProcess[0].difName = "Layer034"

**.host21.ipcProcess0.difName = "Layer021"
**.router2.ipcProcess[1].difName = "Layer021"

**.host22.ipcProcess0.difName = "Layer022"
**.router2.ipcProcess[2].difName = "Layer022"

**.host23.ipcProcess0.difName = "Layer023"
```

```
**.router2.ipcProcess[3].difName = "Layer023"

**.host24.ipcProcess0.difName = "Layer024"
**.router2.ipcProcess[4].difName = "Layer024"

**.host25.ipcProcess0.difName = "Layer025"
**.router2.ipcProcess[5].difName = "Layer025"


#Static addressing: higher IPC layer
**.host11.ipcProcess1.ipcAddress = "111"
**.host12.ipcProcess1.ipcAddress = "112"
**.host13.ipcProcess1.ipcAddress = "113"
**.host14.ipcProcess1.ipcAddress = "114"
**.host15.ipcProcess1.ipcAddress = "115"

**.host21.ipcProcess1.ipcAddress = "121"
**.host22.ipcProcess1.ipcAddress = "122"
**.host23.ipcProcess1.ipcAddress = "123"
**.host24.ipcProcess1.ipcAddress = "124"
**.host25.ipcProcess1.ipcAddress = "125"

**.router1.relayIpc.ipcAddress = "131"
**.router2.relayIpc.ipcAddress = "141"

**.host*.ipcProcess1.difName = "Layer1"
**.router*.relayIpc.difName = "Layer1"

#DIF Allocator settings
**.host11.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host11']/DA")
**.host12.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host12']/DA")
**.host13.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host13']/DA")
**.host14.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host14']/DA")
**.host15.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host15']/DA")
**.host21.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host21']/DA")
**.host22.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host22']/DA")
**.host23.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host23']/DA")
```

```
**.host24.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host24']/DA")
**.host25.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host25']/DA")
#
**.router1.difAllocator.configData = xmldoc("config.xml", "Configuration/
Router[@id='router1']/DA")
**.router2.difAllocator.configData = xmldoc("config.xml", "Configuration/
Router[@id='router2']/DA")
#
##Directory settings
**.host12.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='host11']/DA")
**.host13.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='host11']/DA")
**.host14.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='host11']/DA")
**.host15.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='host11']/DA")
**.host21.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='host11']/DA")
**.host22.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='host11']/DA")
**.host23.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='host11']/DA")
**.host24.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='host11']/DA")
**.host25.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='host11']/DA")
#
**.router2.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Router[@id='router1']/DA")

#
**.ra.qoscubesData = xmldoc("config.xml", "Configuration/QoSCubesSet")

# flows to allocate at the beginning
**.ra.preallocation = \
    xmldoc("config.xml", "Configuration/ConnectionSets/
ConnectionSet[@id='all']/")



[Config Aggregation]
network = SmallNetworkAgg
SmallNetworkAgg.ldelay = 37.5ms
```

```
**.host11.applicationProcess1.applicationEntity.iae.dstApName = "App21"
**.host11.applicationProcess1.applicationEntity.iae.dstAeName = "Stream21"
**.host12.applicationProcess1.applicationEntity.iae.dstApName = "App22"
**.host12.applicationProcess1.applicationEntity.iae.dstAeName = "Stream22"
**.host13.applicationProcess1.applicationEntity.iae.dstApName = "App23"
**.host13.applicationProcess1.applicationEntity.iae.dstAeName = "Stream23"
**.host14.applicationProcess1.applicationEntity.iae.dstApName = "App24"
**.host14.applicationProcess1.applicationEntity.iae.dstAeName = "Stream24"
**.host15.applicationProcess1.applicationEntity.iae.dstApName = "App25"
**.host15.applicationProcess1.applicationEntity.iae.dstAeName = "Stream25"

**.host1*.applicationProcess1.applicationEntity.iae.startAt = 1s
**.host1*.applicationProcess1.applicationEntity.iae.beginStreamAt = 2s
**.host1*.applicationProcess1.applicationEntity.iae.endStreamAt = 162s
**.host1*.applicationProcess1.applicationEntity.iae.interval = 0.002s
**.host1*.applicationProcess1.applicationEntity.iae.stopAt = 162s
**.host1*.applicationProcess1.applicationEntity.iae.size = 536B
**.host1*.applicationProcess1.applicationEntity.iae.forceOrder = true
**.host1*.ipcProcess1.efcp.efcp.txControlPolicy =
 "DTCPTxControlPolicyTCPTahoe"
**.host1*.ipcProcess1.efcp.efcp.rttEstimatorPolicy =
 "DTPRTTEstimatorPolicyTCP"
**.host1*.ipcProcess1.efcp.efcp.senderAckPolicy = "DTCPSenderAckPolicyTCP"
**.host1*.ipcProcess1.efcp.efcp.maxClosedWinQueLen = 200

# Upstream Notification
**.router1.relayIpc.relayAndMux.defaultMaxQLength = 175
**.router1.relayIpc.relayAndMux.defaultThreshQLength = 175
**.router1.relayIpc.relayAndMux.maxQPolicyName = "UpstreamNotifier"

**.router1.ipcProcess[5].relayAndMux.defaultMaxQLength = 175
**.router1.ipcProcess[5].relayAndMux.defaultThreshQLength = 175
**.router1.ipcProcess[5].relayAndMux.maxQPolicyName = "UpstreamNotifier"
**.router1.ipcProcess[5].efcp.efcp.txControlPolicy =
 "DTCPTxControlPolicyTCPTahoe"
**.router1.ipcProcess[5].efcp.efcp.rttEstimatorPolicy =
 "DTPRTTEstimatorPolicyTCP"
**.router1.ipcProcess[5].efcp.efcp.senderAckPolicy =
 "DTCPSenderAckPolicyTCP"
**.router1.ipcProcess[5].efcp.efcp.maxClosedWinQueLen = 25
# End; Upstream Notification

**.host*.ipcProcess*.efcp.efcp.initialSenderCredit = 600
**.host*.ipcProcess*.efcp.efcp.maxClosedWinQueLen = 100000#50000
**.host*.ipcProcess*.efcp.efcp.rcvCredit = 600#122
```

```
**.router*.ipcProcess*.efcp.efcp.initialSenderCredit = 600
**.router*.ipcProcess*.efcp.efcp.maxClosedWinQueLen = 50000
**.router*.ipcProcess*.efcp.efcp.rcvCredit = 600

**.defaultThreshQLength = 50000
**.defaultMaxQLength = 50000
```

8.4.5. config.xml

```
<?xml version="1.0"?>
<Configuration>
    <ConnectionSets>
        <ConnectionSet id="all">
            <SimTime t="0">
                <Connection src="111_Layer1" dst="131_Layer1" qosCube="1"/
>
                <Connection src="112_Layer1" dst="131_Layer1" qosCube="1"/
>
                <Connection src="113_Layer1" dst="131_Layer1" qosCube="1"/
>
                <Connection src="114_Layer1" dst="131_Layer1" qosCube="1"/
>
                <Connection src="115_Layer1" dst="131_Layer1" qosCube="1"/
>

                <Connection src="131_Layer1" dst="141_Layer1" qosCube="1"/
>

                <Connection src="141_Layer1" dst="121_Layer1" qosCube="1"/
>
                <Connection src="141_Layer1" dst="122_Layer1" qosCube="1"/
>
                <Connection src="141_Layer1" dst="123_Layer1" qosCube="1"/
>
                <Connection src="141_Layer1" dst="124_Layer1" qosCube="1"/
>
                <Connection src="141_Layer1" dst="125_Layer1" qosCube="1"/
>
            </SimTime>
        </ConnectionSet>
    </ConnectionSets>

 <Host id="host11">
  <DA>
```

```
<Directory>
 <APN apn="App11">
  <DIF difName="Layer1" ipcAddress="111" />
 </APN>
 <APN apn="App12">
  <DIF difName="Layer1" ipcAddress="112" />
 </APN>
 <APN apn="App13">
  <DIF difName="Layer1" ipcAddress="113" />
 </APN>
 <APN apn="App14">
  <DIF difName="Layer1" ipcAddress="114" />
 </APN>
 <APN apn="App15">
  <DIF difName="Layer1" ipcAddress="115" />
 </APN>
 <APN apn="App21">
  <DIF difName="Layer1" ipcAddress="121" />
 </APN>
 <APN apn="App22">
  <DIF difName="Layer1" ipcAddress="122" />
 </APN>
 <APN apn="App23">
  <DIF difName="Layer1" ipcAddress="123" />
 </APN>
 <APN apn="App24">
  <DIF difName="Layer1" ipcAddress="124" />
 </APN>
 <APN apn="App25">
  <DIF difName="Layer1" ipcAddress="125" />
 </APN>

 <APN apn="111_Layer1">
  <DIF difName="Layer011" ipcAddress="011" />
 </APN>
 <APN apn="112_Layer1">
  <DIF difName="Layer012" ipcAddress="012" />
 </APN>
 <APN apn="113_Layer1">
  <DIF difName="Layer013" ipcAddress="013" />
 </APN>
 <APN apn="114_Layer1">
  <DIF difName="Layer014" ipcAddress="014" />
 </APN>
 <APN apn="115_Layer1">
  <DIF difName="Layer015" ipcAddress="015" />
```

```
    </APN>
    <APN apn="121_Layer1">
     <DIF difName="Layer021" ipcAddress="021" />
    </APN>
    <APN apn="122_Layer1">
     <DIF difName="Layer022" ipcAddress="022" />
    </APN>
    <APN apn="123_Layer1">
     <DIF difName="Layer023" ipcAddress="023" />
    </APN>
    <APN apn="124_Layer1">
     <DIF difName="Layer024" ipcAddress="024" />
    </APN>
    <APN apn="125_Layer1">
     <DIF difName="Layer025" ipcAddress="025" />
    </APN>

    <APN apn="131_Layer1">
     <DIF difName="Layer011" ipcAddress="031" />
     <DIF difName="Layer012" ipcAddress="032" />
     <DIF difName="Layer013" ipcAddress="033" />
     <DIF difName="Layer014" ipcAddress="034" />
     <DIF difName="Layer015" ipcAddress="035" />
     <DIF difName="Layer034" ipcAddress="036" />
    </APN>
    <APN apn="141_Layer1">
     <DIF difName="Layer021" ipcAddress="041" />
     <DIF difName="Layer022" ipcAddress="042" />
     <DIF difName="Layer023" ipcAddress="043" />
     <DIF difName="Layer024" ipcAddress="044" />
     <DIF difName="Layer025" ipcAddress="045" />
     <DIF difName="Layer034" ipcAddress="046" />
    </APN>
   </Directory>
   <NeighborTable>
    <APN apn="121_Layer1">
     <Neighbor apn="131_Layer1" />
    </APN>
   </NeighborTable>
  </DA>
 </Host>

 <Host id="host12">
  <DA>
   <NeighborTable>
    <APN apn="122_Layer1">
```

```
    <Neighbor apn="131_Layer1" />
   </APN>
  </NeighborTable>
 </DA>
</Host>

<Host id="host13">
 <DA>
  <NeighborTable>
   <APN apn="123_Layer1">
    <Neighbor apn="131_Layer1" />
   </APN>
  </NeighborTable>
 </DA>
</Host>

<Host id="host14">
 <DA>
  <NeighborTable>
   <APN apn="124_Layer1">
    <Neighbor apn="131_Layer1" />
   </APN>
  </NeighborTable>
 </DA>
</Host>

<Host id="host15">
 <DA>
  <NeighborTable>
   <APN apn="125_Layer1">
    <Neighbor apn="131_Layer1" />
   </APN>
  </NeighborTable>
 </DA>
</Host>

<Host id="host21">
 <DA>
  <NeighborTable>
   <APN apn="111_Layer1">
    <Neighbor apn="141_Layer1" />
   </APN>
  </NeighborTable>
 </DA>
</Host>
```

```
<Host id="host22">
 <DA>
  <NeighborTable>
   <APN apn="112_Layer1">
    <Neighbor apn="141_Layer1" />
   </APN>
  </NeighborTable>
 </DA>
</Host>

<Host id="host23">
 <DA>
  <NeighborTable>
   <APN apn="113_Layer1">
    <Neighbor apn="141_Layer1" />
   </APN>
  </NeighborTable>
 </DA>
</Host>

<Host id="host24">
 <DA>
  <NeighborTable>
   <APN apn="114_Layer1">
    <Neighbor apn="141_Layer1" />
   </APN>
  </NeighborTable>
 </DA>
</Host>

<Host id="host25">
 <DA>
  <NeighborTable>
   <APN apn="115_Layer1">
    <Neighbor apn="141_Layer1" />
   </APN>
  </NeighborTable>
 </DA>
</Host>

<Router id="router1">
 <DA>
  <Directory>
   <APN apn="App11">
    <DIF difName="Layer1" ipcAddress="111" />
   </APN>
```

```
<APN apn="App12">
 <DIF difName="Layer1" ipcAddress="112" />
</APN>
<APN apn="App13">
 <DIF difName="Layer1" ipcAddress="113" />
</APN>
<APN apn="App14">
 <DIF difName="Layer1" ipcAddress="114" />
</APN>
<APN apn="App15">
 <DIF difName="Layer1" ipcAddress="115" />
</APN>
<APN apn="App21">
 <DIF difName="Layer1" ipcAddress="121" />
</APN>
<APN apn="App22">
 <DIF difName="Layer1" ipcAddress="122" />
</APN>
<APN apn="App23">
 <DIF difName="Layer1" ipcAddress="123" />
</APN>
<APN apn="App24">
 <DIF difName="Layer1" ipcAddress="124" />
</APN>
<APN apn="App25">
 <DIF difName="Layer1" ipcAddress="125" />
</APN>

<APN apn="111_Layer1">
 <DIF difName="Layer011" ipcAddress="011" />
</APN>
<APN apn="112_Layer1">
 <DIF difName="Layer012" ipcAddress="012" />
</APN>
<APN apn="113_Layer1">
 <DIF difName="Layer013" ipcAddress="013" />
</APN>
<APN apn="114_Layer1">
 <DIF difName="Layer014" ipcAddress="014" />
</APN>
<APN apn="115_Layer1">
 <DIF difName="Layer015" ipcAddress="015" />
</APN>
<APN apn="121_Layer1">
 <DIF difName="Layer021" ipcAddress="021" />
</APN>
```

```
 <APN apn="122_Layer1">
  <DIF difName="Layer022" ipcAddress="022" />
 </APN>
 <APN apn="123_Layer1">
  <DIF difName="Layer023" ipcAddress="023" />
 </APN>
 <APN apn="124_Layer1">
  <DIF difName="Layer024" ipcAddress="024" />
 </APN>
 <APN apn="125_Layer1">
  <DIF difName="Layer025" ipcAddress="025" />
 </APN>

 <APN apn="131_Layer1">
  <DIF difName="Layer011" ipcAddress="031" />
  <DIF difName="Layer012" ipcAddress="032" />
  <DIF difName="Layer013" ipcAddress="033" />
  <DIF difName="Layer014" ipcAddress="034" />
  <DIF difName="Layer015" ipcAddress="035" />
  <DIF difName="Layer034" ipcAddress="036" />
 </APN>
 <APN apn="141_Layer1">
  <DIF difName="Layer021" ipcAddress="041" />
  <DIF difName="Layer022" ipcAddress="042" />
  <DIF difName="Layer023" ipcAddress="043" />
  <DIF difName="Layer024" ipcAddress="044" />
  <DIF difName="Layer025" ipcAddress="045" />
  <DIF difName="Layer034" ipcAddress="046" />
 </APN>
</Directory>
<NeighborTable>
 <APN apn="121_Layer1">
  <Neighbor apn="141_Layer1" />
 </APN>
 <APN apn="122_Layer1">
  <Neighbor apn="141_Layer1" />
 </APN>
 <APN apn="123_Layer1">
  <Neighbor apn="141_Layer1" />
 </APN>
 <APN apn="124_Layer1">
  <Neighbor apn="141_Layer1" />
 </APN>
 <APN apn="125_Layer1">
  <Neighbor apn="141_Layer1" />
 </APN>
```

```
  </NeighborTable>
 </DA>
</Router>
<Router id="router2">
 <DA>
  <NeighborTable>
   <APN apn="111_Layer1">
    <Neighbor apn="131_Layer1" />
   </APN>
   <APN apn="112_Layer1">
    <Neighbor apn="131_Layer1" />
   </APN>
   <APN apn="113_Layer1">
    <Neighbor apn="131_Layer1" />
   </APN>
   <APN apn="114_Layer1">
    <Neighbor apn="131_Layer1" />
   </APN>
   <APN apn="115_Layer1">
    <Neighbor apn="131_Layer1" />
   </APN>
  </NeighborTable>
 </DA>
</Router>

<QoSCubesSet>
       <QosCube id="1">
           <AverageBandwidth>12000000</AverageBandwidth>
           <AverageSDUBandwidth>1000</AverageSDUBandwidth>
           <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
           <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
           <BurstPeriod>10000000</BurstPeriod>
           <BurstDuration>1000000</BurstDuration>
           <UndetectedBitError>0.01</UndetectedBitError>
           <MaxSDUSize>1500</MaxSDUSize>
           <PartialDelivery>0</PartialDelivery>
           <IncompleteDelivery>0</IncompleteDelivery>
           <ForceOrder>1</ForceOrder>
           <MaxAllowableGap>0</MaxAllowableGap>
           <Delay>1000000</Delay>
           <Jitter>500000</Jitter>
           <CostTime>0</CostTime>
           <CostBits>0</CostBits>
           <ATime>0</ATime>
       </QosCube>
</QoSCubesSet>
```

```
</Configuration>
```

8.5. Demonstration: Routing

8.5.1. Motivation

The goal of this demonstration is to show how RINA's Forwarding, PDUGE and Routing policies are used in a simple network topology with distinct requirements.

8.5.2. Description

To achieve this goal, we simulated a scenario in which there were multiple paths between hosts with distinct properties. The example is named LatEx in the *example/Routing/LatEx* folder of RINASim. The network topology is shown in Figure 67. host1 communicates with host2. The links between Routers_2 and Router_4 and Routers_3 and Router_4 have various length, forming the path with different latencies, given by the shim DIFs QoS Cubes.

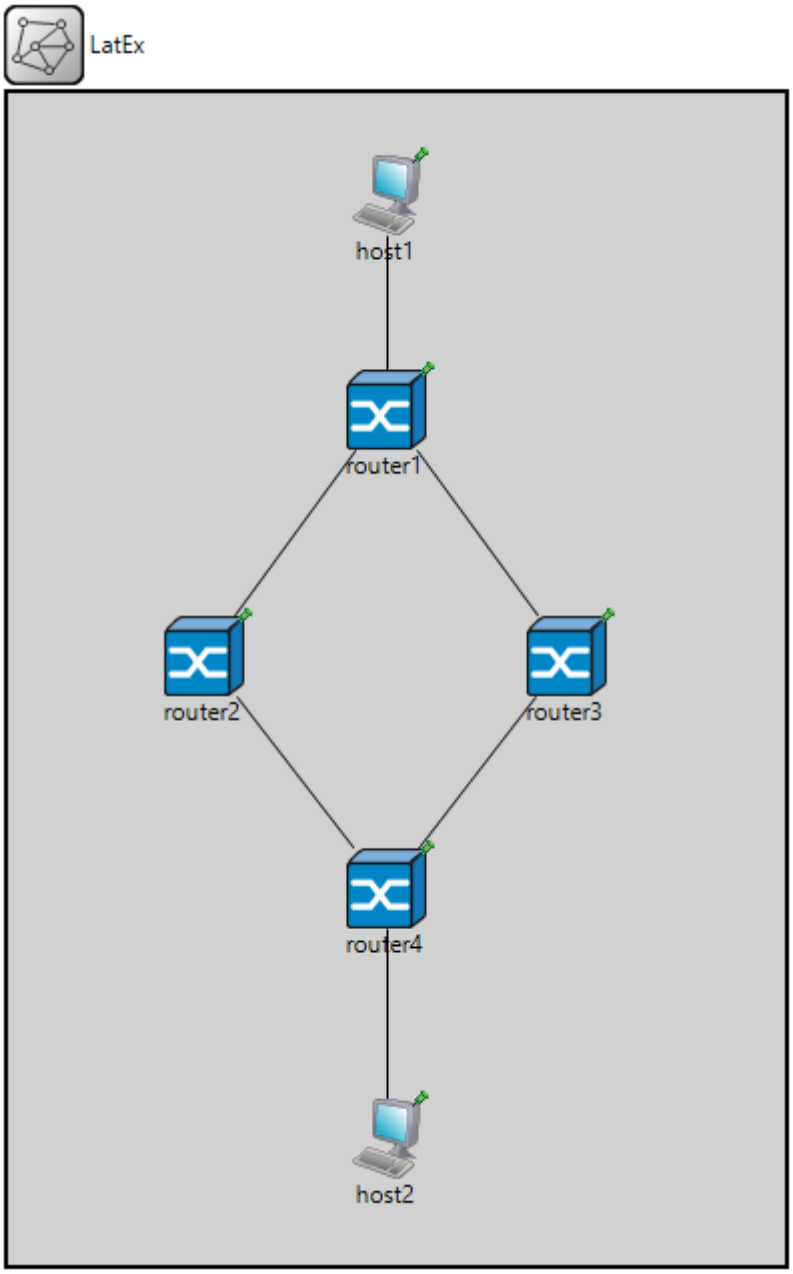

Figure 67. Network topology

8.5.3. Configurations

The example is used to test multiple routing policies and has multiple configurations with individual results.

- Hop*, HR Forwarding using the minimum length paths to a destination. Some use memoryless ECMP if more than one path is found.
- Lat* Forwarding using the minimum latency paths to a destination. Some use memoryless ECMP if more than one path is found.

8.5.4. omnetpp.ini

```
[General]
network = LatEx
sim-time-limit =  5min

**.host1.**.ipcAddress = "h1"
**.host2.**.ipcAddress = "h2"

**.router1.**.ipcAddress = "r1"
**.router2.**.ipcAddress = "r2"
**.router3.**.ipcAddress = "r3"
**.router4.**.ipcAddress = "r4"


**.host*.ipcProcess1.difName = "NET"
**.router*.relayIpc.difName = "NET"

**.host1.ipcProcess0.difName = "shimHR1"
**.router1.ipcProcess[0].difName = "shimHR1"

**.host2.ipcProcess0.difName = "shimHR2"
**.router4.ipcProcess[0].difName = "shimHR2"

**.router1.ipcProcess[1].difName = "shim12"
**.router2.ipcProcess[0].difName = "shim12"

**.router1.ipcProcess[2].difName = "shim13"
**.router3.ipcProcess[0].difName = "shim13"

**.router4.ipcProcess[1].difName = "shim24"
**.router2.ipcProcess[1].difName = "shim24"

**.router4.ipcProcess[2].difName = "shim34"
**.router3.ipcProcess[1].difName = "shim34"


**.flowAllocator.newFlowReqPolicyType = "MinComparer"

**.ra.qoscubesData = xmldoc("QoS.xml", "Configuration/QoSCubesSet")
**.ra.qosReqData = xmldoc("QoS.xml", "Configuration/QoSReqSet")

**.ra.preallocation = xmldoc("connections.xml", "Configuration/
ConnectionSet")
```

```
**.difAllocator.configData = xmldoc("config.xml", "Configuration/DA")
**.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/DA")


**.relayIpc.**.pduForwardingPolicy.printAtEnd = true
**.ipcProcess1.**.pduForwardingPolicy.printAtEnd = true
**.relayIpc.routingPolicy.printAtEnd = true
**.ipcProcess1.routingPolicy.printAtEnd = true
**.ipcProcess1.**.printAtEnd = true
**.printAtEnd = false

**.ipcProcess1.relayAndMux.ForwardingPolicyName = "MiniTable"
**.relayIpc.relayAndMux.ForwardingPolicyName = "MiniTable"

#
# Appliction entities naming:
#
**.host1.applicationProcess1.apName = "Snd"
**.host2.applicationProcess1.apName = "Rcv"

**.applicationEntity.aeType    = "AEPing"
**.iae.aeName        = "Ping"

**.host1.applicationProcess1.applicationEntity.iae.dstApName = "Rcv"
**.host1.applicationProcess1.applicationEntity.iae.dstAeName  = "Ping"
**.host1.applicationProcess1.applicationEntity.iae.startAt   = 130s
**.host1.applicationProcess1.applicationEntity.iae.pingAt   = 140s
**.host1.applicationProcess1.applicationEntity.iae.rate   = 5
**.host1.applicationProcess1.applicationEntity.iae.stopAt   = 0


[Config HopDV]

**.ipcProcess1.resourceAllocator.pdufgPolicyName = "SimpleGenerator"
**.relayIpc.resourceAllocator.pdufgPolicyName = "SimpleGenerator"

**.ipcProcess1.routingPolicyName = "SimpleDV"
**.relayIpc.routingPolicyName = "SimpleDV"


[Config HopLS]

**.ipcProcess1.resourceAllocator.pdufgPolicyName = "SimpleGenerator"
**.relayIpc.resourceAllocator.pdufgPolicyName = "SimpleGenerator"
```

```
**.ipcProcess1.routingPolicyName = "SimpleLS"
**.relayIpc.routingPolicyName = "SimpleLS"



[Config LatDV]


**.ipcProcess1.routingPolicy.infMetric = 1000
**.relayIpc.routingPolicy.infMetric = 1000

**.ipcProcess1.resourceAllocator.pdufgPolicyName = "LatGenerator"
**.relayIpc.resourceAllocator.pdufgPolicyName = "LatGenerator"

**.ipcProcess1.routingPolicyName = "SimpleDV"
**.relayIpc.routingPolicyName = "SimpleDV"


[Config LatLS]

**.ipcProcess1.resourceAllocator.pdufgPolicyName = "LatGenerator"
**.relayIpc.resourceAllocator.pdufgPolicyName = "LatGenerator"

**.ipcProcess1.routingPolicyName = "SimpleLS"
**.relayIpc.routingPolicyName = "SimpleLS"

[Config HopsSingle1EntryLS]

**.ipcProcess1.resourceAllocator.pdufgPolicyName = "HopsSingle1Entry"
**.relayIpc.resourceAllocator.pdufgPolicyName = "HopsSingle1Entry"

**.ipcProcess1.routingPolicyName = "TSimpleLS"
**.relayIpc.routingPolicyName = "TSimpleLS"

[Config HopsSingleMEntriesLS]

**.ipcProcess1.resourceAllocator.pdufgPolicyName = "HopsSingleMEntries"
**.relayIpc.resourceAllocator.pdufgPolicyName = "HopsSingleMEntries"

**.ipcProcess1.routingPolicyName = "TSimpleLS"
**.relayIpc.routingPolicyName = "TSimpleLS"

**.ipcProcess1.relayAndMux.ForwardingPolicyName = "MultiMiniTable"
**.relayIpc.relayAndMux.ForwardingPolicyName = "MultiMiniTable"
```

```
[Config LatencySingle1EntryLS]

**.ipcProcess1.resourceAllocator.pdufgPolicyName = "LatencySingle1Entry"
**.relayIpc.resourceAllocator.pdufgPolicyName = "LatencySingle1Entry"

**.ipcProcess1.routingPolicyName = "TSimpleLS"
**.relayIpc.routingPolicyName = "TSimpleLS"


[Config LatencySingleMEntriesLS]

**.ipcProcess1.resourceAllocator.pdufgPolicyName = "LatencySingleMEntries"
**.relayIpc.resourceAllocator.pdufgPolicyName = "LatencySingleMEntries"

**.ipcProcess1.routingPolicyName = "TSimpleLS"
**.relayIpc.routingPolicyName = "TSimpleLS"

**.ipcProcess1.relayAndMux.ForwardingPolicyName = "MultiMiniTable"
**.relayIpc.relayAndMux.ForwardingPolicyName = "MultiMiniTable"



[Config HopsSingle1EntryDV]

**.ipcProcess1.resourceAllocator.pdufgPolicyName = "HopsSingle1Entry"
**.relayIpc.resourceAllocator.pdufgPolicyName = "HopsSingle1Entry"

**.ipcProcess1.routingPolicyName = "TSimpleDV"
**.relayIpc.routingPolicyName = "TSimpleDV"

[Config HopsSingleMEntriesDV]

**.ipcProcess1.resourceAllocator.pdufgPolicyName = "HopsSingleMEntries"
**.relayIpc.resourceAllocator.pdufgPolicyName = "HopsSingleMEntries"

**.ipcProcess1.routingPolicyName = "TSimpleDV"
**.relayIpc.routingPolicyName = "TSimpleDV"

**.ipcProcess1.relayAndMux.ForwardingPolicyName = "MultiMiniTable"
**.relayIpc.relayAndMux.ForwardingPolicyName = "MultiMiniTable"



[Config LatencySingle1EntryDV]
```

```
**.ipcProcess1.resourceAllocator.pdufgPolicyName = "LatencySingle1Entry"
**.relayIpc.resourceAllocator.pdufgPolicyName = "LatencySingle1Entry"

**.ipcProcess1.routingPolicyName = "TSimpleDV"
**.relayIpc.routingPolicyName = "TSimpleDV"


[Config LatencySingleMEntriesDV]

**.ipcProcess1.resourceAllocator.pdufgPolicyName = "LatencySingleMEntries"
**.relayIpc.resourceAllocator.pdufgPolicyName = "LatencySingleMEntries"

**.ipcProcess1.routingPolicyName = "TSimpleDV"
**.relayIpc.routingPolicyName = "TSimpleDV"

**.ipcProcess1.relayAndMux.ForwardingPolicyName = "MultiMiniTable"
**.relayIpc.relayAndMux.ForwardingPolicyName = "MultiMiniTable"


[Config HR]
**.relayIpc.resourceAllocator.pdufgPolicyName = "HierarchicalGenerator"
**.relayIpc.routingPolicyName = "TDomainRouting"
**.relayIpc.relayAndMux.ForwardingPolicyName = "0"

**.ipcProcess1.resourceAllocator.pdufgPolicyName = "HierarchicalGenerator"
**.ipcProcess1.routingPolicyName = "TDomainRouting"
**.ipcProcess1.relayAndMux.ForwardingPolicyName = "HierarchicalTable"
```

8.5.5. config.xml

```
<?xml version="1.0"?>
<Configuration>
  <DA>
    <Directory>
  <APN apn="h1_NET">
   <DIF difName="shimHR1" ipcAddress="h1" />
  </APN>
  <APN apn="h2_NET">
   <DIF difName="shimHR2" ipcAddress="h2" />
  </APN>

  <APN apn="r1_NET">
   <DIF difName="shimHR1" ipcAddress="r1" />
   <DIF difName="shim12" ipcAddress="r1" />
   <DIF difName="shim13" ipcAddress="r1" />
```

```
  </APN>
  <APN apn="r2_NET">
   <DIF difName="shim12" ipcAddress="r2" />
   <DIF difName="shim24" ipcAddress="r2" />
  </APN>
  <APN apn="r3_NET">
   <DIF difName="shim13" ipcAddress="r3" />
   <DIF difName="shim34" ipcAddress="r3" />
  </APN>
  <APN apn="r4_NET">
   <DIF difName="shimHR2" ipcAddress="r4" />
   <DIF difName="shim24" ipcAddress="r4" />
   <DIF difName="shim34" ipcAddress="r4" />
  </APN>


  <APN apn="Snd">
   <DIF difName="NET" ipcAddress="h1" />
  </APN>
  <APN apn="Rcv">
   <DIF difName="NET" ipcAddress="h2" />
  </APN>
    </Directory>
  </DA>
</Configuration>
```

8.5.6. QoS.xml

```
<?xml version="1.0"?>
<Configuration>
 <QoSReqSet>
  <QosReq id="1">
   <Delay>1</Delay>
  </QosReq>
  <QosReq id="2">
   <Delay>2</Delay>
  </QosReq>
  <QosReq id="3">
   <Delay>3</Delay>
  </QosReq>
  <QosReq id="4">
   <Delay>4</Delay>
  </QosReq>
  <QosReq id="5">
   <Delay>5</Delay>
```

```
 </QosReq>
 <QosReq id="6">
  <Delay>6</Delay>
 </QosReq>
 <QosReq id="7">
  <Delay>7</Delay>
 </QosReq>
 <QosReq id="8">
  <Delay>8</Delay>
 </QosReq>
 <QosReq id="9">
  <Delay>9</Delay>
 </QosReq>
 <QosReq id="10">
  <Delay>10</Delay>
 </QosReq>
 <QosReq id="15">
  <Delay>15</Delay>
 </QosReq>
 <QosReq id="20">
  <Delay>20</Delay>
 </QosReq>
 <QosReq id="30">
  <Delay>30</Delay>
 </QosReq>
 <QosReq id="50">
  <Delay>50</Delay>
 </QosReq>
 <QosReq id="100">
  <Delay>100</Delay>
 </QosReq>
</QoSReqSet>

<QoSCubesSet>
 <QosCube id="1">
  <Delay>1</Delay>
  <CostBits>1</CostBits>


  <PDUDroppingProbability>0.001</PDUDroppingProbability>
  <CostTime>10000</CostTime>
  <AverageBandwidth>12000000</AverageBandwidth>
  <AverageSDUBandwidth>1000</AverageSDUBandwidth>
  <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
  <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
  <BurstPeriod>10000000</BurstPeriod>
```

```
 <BurstDuration>1000000</BurstDuration>
 <UndetectedBitError>0.01</UndetectedBitError>
 <MaxSDUSize>1500</MaxSDUSize>
 <PartialDelivery>0</PartialDelivery>
 <IncompleteDelivery>0</IncompleteDelivery>
 <ForceOrder>0</ForceOrder>
 <MaxAllowableGap>0</MaxAllowableGap>
 <Jitter>500000</Jitter>
 <ATime>0</ATime>
 <RxOn>0</RxOn>
 <WinOn>0</WinOn>
 <RateOn>0</RateOn>
</QosCube>

<QosCube id="2">
 <Delay>2</Delay>
 <CostBits>2</CostBits>


 <PDUDroppingProbability>0.001</PDUDroppingProbability>
 <CostTime>10000</CostTime>
 <AverageBandwidth>12000000</AverageBandwidth>
 <AverageSDUBandwidth>1000</AverageSDUBandwidth>
 <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
 <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
 <BurstPeriod>10000000</BurstPeriod>
 <BurstDuration>1000000</BurstDuration>
 <UndetectedBitError>0.01</UndetectedBitError>
 <MaxSDUSize>1500</MaxSDUSize>
 <PartialDelivery>0</PartialDelivery>
 <IncompleteDelivery>0</IncompleteDelivery>
 <ForceOrder>0</ForceOrder>
 <MaxAllowableGap>0</MaxAllowableGap>
 <Jitter>500000</Jitter>
 <ATime>0</ATime>
 <RxOn>0</RxOn>
 <WinOn>0</WinOn>
 <RateOn>0</RateOn>
</QosCube>

<QosCube id="3">
 <Delay>3</Delay>
 <CostBits>3</CostBits>


 <PDUDroppingProbability>0.001</PDUDroppingProbability>
```

```
 <CostTime>10000</CostTime>
 <AverageBandwidth>12000000</AverageBandwidth>
 <AverageSDUBandwidth>1000</AverageSDUBandwidth>
 <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
 <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
 <BurstPeriod>10000000</BurstPeriod>
 <BurstDuration>1000000</BurstDuration>
 <UndetectedBitError>0.01</UndetectedBitError>
 <MaxSDUSize>1500</MaxSDUSize>
 <PartialDelivery>0</PartialDelivery>
 <IncompleteDelivery>0</IncompleteDelivery>
 <ForceOrder>0</ForceOrder>
 <MaxAllowableGap>0</MaxAllowableGap>
 <Jitter>500000</Jitter>
 <ATime>0</ATime>
 <RxOn>0</RxOn>
 <WinOn>0</WinOn>
 <RateOn>0</RateOn>
</QosCube>

<QosCube id="4">
 <Delay>4</Delay>
 <CostBits>4</CostBits>


 <PDUDroppingProbability>0.001</PDUDroppingProbability>
 <CostTime>10000</CostTime>
 <AverageBandwidth>12000000</AverageBandwidth>
 <AverageSDUBandwidth>1000</AverageSDUBandwidth>
 <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
 <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
 <BurstPeriod>10000000</BurstPeriod>
 <BurstDuration>1000000</BurstDuration>
 <UndetectedBitError>0.01</UndetectedBitError>
 <MaxSDUSize>1500</MaxSDUSize>
 <PartialDelivery>0</PartialDelivery>
 <IncompleteDelivery>0</IncompleteDelivery>
 <ForceOrder>0</ForceOrder>
 <MaxAllowableGap>0</MaxAllowableGap>
 <Jitter>500000</Jitter>
 <ATime>0</ATime>
 <RxOn>0</RxOn>
 <WinOn>0</WinOn>
 <RateOn>0</RateOn>
</QosCube>
```

```
<QosCube id="5">
 <Delay>5</Delay>
 <CostBits>5</CostBits>


 <PDUDroppingProbability>0.001</PDUDroppingProbability>
 <CostTime>10000</CostTime>
 <AverageBandwidth>12000000</AverageBandwidth>
 <AverageSDUBandwidth>1000</AverageSDUBandwidth>
 <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
 <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
 <BurstPeriod>10000000</BurstPeriod>
 <BurstDuration>1000000</BurstDuration>
 <UndetectedBitError>0.01</UndetectedBitError>
 <MaxSDUSize>1500</MaxSDUSize>
 <PartialDelivery>0</PartialDelivery>
 <IncompleteDelivery>0</IncompleteDelivery>
 <ForceOrder>0</ForceOrder>
 <MaxAllowableGap>0</MaxAllowableGap>
 <Jitter>500000</Jitter>
 <ATime>0</ATime>
 <RxOn>0</RxOn>
 <WinOn>0</WinOn>
 <RateOn>0</RateOn>
</QosCube>

<QosCube id="6">
 <Delay>6</Delay>
 <CostBits>6</CostBits>


 <PDUDroppingProbability>0.001</PDUDroppingProbability>
 <CostTime>10000</CostTime>
 <AverageBandwidth>12000000</AverageBandwidth>
 <AverageSDUBandwidth>1000</AverageSDUBandwidth>
 <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
 <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
 <BurstPeriod>10000000</BurstPeriod>
 <BurstDuration>1000000</BurstDuration>
 <UndetectedBitError>0.01</UndetectedBitError>
 <MaxSDUSize>1500</MaxSDUSize>
 <PartialDelivery>0</PartialDelivery>
 <IncompleteDelivery>0</IncompleteDelivery>
 <ForceOrder>0</ForceOrder>
 <MaxAllowableGap>0</MaxAllowableGap>
 <Jitter>500000</Jitter>
```

```
 <ATime>0</ATime>
 <RxOn>0</RxOn>
 <WinOn>0</WinOn>
 <RateOn>0</RateOn>
</QosCube>

<QosCube id="7">
 <Delay>7</Delay>
 <CostBits>7</CostBits>


 <PDUDroppingProbability>0.001</PDUDroppingProbability>
 <CostTime>10000</CostTime>
 <AverageBandwidth>12000000</AverageBandwidth>
 <AverageSDUBandwidth>1000</AverageSDUBandwidth>
 <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
 <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
 <BurstPeriod>10000000</BurstPeriod>
 <BurstDuration>1000000</BurstDuration>
 <UndetectedBitError>0.01</UndetectedBitError>
 <MaxSDUSize>1500</MaxSDUSize>
 <PartialDelivery>0</PartialDelivery>
 <IncompleteDelivery>0</IncompleteDelivery>
 <ForceOrder>0</ForceOrder>
 <MaxAllowableGap>0</MaxAllowableGap>
 <Jitter>500000</Jitter>
 <ATime>0</ATime>
 <RxOn>0</RxOn>
 <WinOn>0</WinOn>
 <RateOn>0</RateOn>
</QosCube>

<QosCube id="8">
 <Delay>8</Delay>
 <CostBits>8</CostBits>


 <PDUDroppingProbability>0.001</PDUDroppingProbability>
 <CostTime>10000</CostTime>
 <AverageBandwidth>12000000</AverageBandwidth>
 <AverageSDUBandwidth>1000</AverageSDUBandwidth>
 <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
 <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
 <BurstPeriod>10000000</BurstPeriod>
 <BurstDuration>1000000</BurstDuration>
 <UndetectedBitError>0.01</UndetectedBitError>
```

```
 <MaxSDUSize>1500</MaxSDUSize>
 <PartialDelivery>0</PartialDelivery>
 <IncompleteDelivery>0</IncompleteDelivery>
 <ForceOrder>0</ForceOrder>
 <MaxAllowableGap>0</MaxAllowableGap>
 <Jitter>500000</Jitter>
 <ATime>0</ATime>
 <RxOn>0</RxOn>
 <WinOn>0</WinOn>
 <RateOn>0</RateOn>
</QosCube>

<QosCube id="9">
 <Delay>9</Delay>
 <CostBits>9</CostBits>


 <PDUDroppingProbability>0.001</PDUDroppingProbability>
 <CostTime>10000</CostTime>
 <AverageBandwidth>12000000</AverageBandwidth>
 <AverageSDUBandwidth>1000</AverageSDUBandwidth>
 <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
 <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
 <BurstPeriod>10000000</BurstPeriod>
 <BurstDuration>1000000</BurstDuration>
 <UndetectedBitError>0.01</UndetectedBitError>
 <MaxSDUSize>1500</MaxSDUSize>
 <PartialDelivery>0</PartialDelivery>
 <IncompleteDelivery>0</IncompleteDelivery>
 <ForceOrder>0</ForceOrder>
 <MaxAllowableGap>0</MaxAllowableGap>
 <Jitter>500000</Jitter>
 <ATime>0</ATime>
 <RxOn>0</RxOn>
 <WinOn>0</WinOn>
 <RateOn>0</RateOn>
</QosCube>

<QosCube id="10">
 <Delay>10</Delay>
 <CostBits>10</CostBits>


 <PDUDroppingProbability>0.001</PDUDroppingProbability>
 <CostTime>10000</CostTime>
 <AverageBandwidth>12000000</AverageBandwidth>
```

```
 <AverageSDUBandwidth>1000</AverageSDUBandwidth>
 <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
 <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
 <BurstPeriod>10000000</BurstPeriod>
 <BurstDuration>1000000</BurstDuration>
 <UndetectedBitError>0.01</UndetectedBitError>
 <MaxSDUSize>1500</MaxSDUSize>
 <PartialDelivery>0</PartialDelivery>
 <IncompleteDelivery>0</IncompleteDelivery>
 <ForceOrder>0</ForceOrder>
 <MaxAllowableGap>0</MaxAllowableGap>
 <Jitter>500000</Jitter>
 <ATime>0</ATime>
 <RxOn>0</RxOn>
 <WinOn>0</WinOn>
 <RateOn>0</RateOn>
</QosCube>

<QosCube id="15">
 <Delay>15</Delay>
 <CostBits>15</CostBits>


 <PDUDroppingProbability>0.001</PDUDroppingProbability>
 <CostTime>10000</CostTime>
 <AverageBandwidth>12000000</AverageBandwidth>
 <AverageSDUBandwidth>1000</AverageSDUBandwidth>
 <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
 <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
 <BurstPeriod>10000000</BurstPeriod>
 <BurstDuration>1000000</BurstDuration>
 <UndetectedBitError>0.01</UndetectedBitError>
 <MaxSDUSize>1500</MaxSDUSize>
 <PartialDelivery>0</PartialDelivery>
 <IncompleteDelivery>0</IncompleteDelivery>
 <ForceOrder>0</ForceOrder>
 <MaxAllowableGap>0</MaxAllowableGap>
 <Jitter>500000</Jitter>
 <ATime>0</ATime>
 <RxOn>0</RxOn>
 <WinOn>0</WinOn>
 <RateOn>0</RateOn>
</QosCube>

<QosCube id="20">
 <Delay>20</Delay>
```

```
  <CostBits>20</CostBits>


  <PDUDroppingProbability>0.001</PDUDroppingProbability>
  <CostTime>10000</CostTime>
  <AverageBandwidth>12000000</AverageBandwidth>
  <AverageSDUBandwidth>1000</AverageSDUBandwidth>
  <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
  <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
  <BurstPeriod>10000000</BurstPeriod>
  <BurstDuration>1000000</BurstDuration>
  <UndetectedBitError>0.01</UndetectedBitError>
  <MaxSDUSize>1500</MaxSDUSize>
  <PartialDelivery>0</PartialDelivery>
  <IncompleteDelivery>0</IncompleteDelivery>
  <ForceOrder>0</ForceOrder>
  <MaxAllowableGap>0</MaxAllowableGap>
  <Jitter>500000</Jitter>
  <ATime>0</ATime>
  <RxOn>0</RxOn>
  <WinOn>0</WinOn>
  <RateOn>0</RateOn>
 </QosCube>

 <QosCube id="30">
  <Delay>30</Delay>
  <CostBits>30</CostBits>


  <PDUDroppingProbability>0.001</PDUDroppingProbability>
  <CostTime>10000</CostTime>
  <AverageBandwidth>12000000</AverageBandwidth>
  <AverageSDUBandwidth>1000</AverageSDUBandwidth>
  <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
  <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
  <BurstPeriod>10000000</BurstPeriod>
  <BurstDuration>1000000</BurstDuration>
  <UndetectedBitError>0.01</UndetectedBitError>
  <MaxSDUSize>1500</MaxSDUSize>
  <PartialDelivery>0</PartialDelivery>
  <IncompleteDelivery>0</IncompleteDelivery>
  <ForceOrder>0</ForceOrder>
  <MaxAllowableGap>0</MaxAllowableGap>
  <Jitter>500000</Jitter>
  <ATime>0</ATime>
  <RxOn>0</RxOn>
```

```
 <WinOn>0</WinOn>
 <RateOn>0</RateOn>
</QosCube>

<QosCube id="50">
 <Delay>50</Delay>
 <CostBits>50</CostBits>


 <PDUDroppingProbability>0.001</PDUDroppingProbability>
 <CostTime>10000</CostTime>
 <AverageBandwidth>12000000</AverageBandwidth>
 <AverageSDUBandwidth>1000</AverageSDUBandwidth>
 <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
 <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
 <BurstPeriod>10000000</BurstPeriod>
 <BurstDuration>1000000</BurstDuration>
 <UndetectedBitError>0.01</UndetectedBitError>
 <MaxSDUSize>1500</MaxSDUSize>
 <PartialDelivery>0</PartialDelivery>
 <IncompleteDelivery>0</IncompleteDelivery>
 <ForceOrder>0</ForceOrder>
 <MaxAllowableGap>0</MaxAllowableGap>
 <Jitter>500000</Jitter>
 <ATime>0</ATime>
 <RxOn>0</RxOn>
 <WinOn>0</WinOn>
 <RateOn>0</RateOn>
</QosCube>

<QosCube id="100">
 <Delay>100</Delay>
 <CostBits>100</CostBits>


 <PDUDroppingProbability>0.001</PDUDroppingProbability>
 <CostTime>10000</CostTime>
 <AverageBandwidth>12000000</AverageBandwidth>
 <AverageSDUBandwidth>1000</AverageSDUBandwidth>
 <PeakBandwidthDuration>24000000</PeakBandwidthDuration>
 <PeakSDUBandwidthDuration>2000</PeakSDUBandwidthDuration>
 <BurstPeriod>10000000</BurstPeriod>
 <BurstDuration>1000000</BurstDuration>
 <UndetectedBitError>0.01</UndetectedBitError>
 <MaxSDUSize>1500</MaxSDUSize>
 <PartialDelivery>0</PartialDelivery>
```

```
    <IncompleteDelivery>0</IncompleteDelivery>
    <ForceOrder>0</ForceOrder>
    <MaxAllowableGap>0</MaxAllowableGap>
    <Jitter>500000</Jitter>
    <ATime>0</ATime>
    <RxOn>0</RxOn>
    <WinOn>0</WinOn>
    <RateOn>0</RateOn>
   </QosCube>
  </QoSCubesSet>
</Configuration>
```

8.5.7. connections.xml

```
<?xml version="1.0"?>
<Configuration>
   <ConnectionSet>
    <!-- Allocate management flows. -->
    <SimTime t="1">
     <Connection src="h1_NET" dst="r1_NET" qosReq="mgmt"/>
     <Connection src="r2_NET" dst="r1_NET" qosReq="mgmt"/>
     <Connection src="r3_NET" dst="r1_NET" qosReq="mgmt"/>
     <Connection src="h2_NET" dst="r4_NET" qosReq="mgmt"/>
     <Connection src="r2_NET" dst="r4_NET" qosReq="mgmt"/>
     <Connection src="r3_NET" dst="r4_NET" qosReq="mgmt"/>
    </SimTime>

        <!-- Allocate data flows. -->
    <SimTime t="2">
     <Connection src="h1_NET" dst="r1_NET" qosReq="1"/>
     <Connection src="r2_NET" dst="r1_NET" qosReq="1"/>
     <Connection src="r3_NET" dst="r1_NET" qosReq="1"/>
     <Connection src="h2_NET" dst="r4_NET" qosReq="1"/>
     <Connection src="r2_NET" dst="r4_NET" qosReq="10"/>
     <Connection src="r3_NET" dst="r4_NET" qosReq="15"/>
    </SimTime>
  </ConnectionSet>
</Configuration>
```

9. Conclusions

The presented deliverable documents the advances in implementation fo RINASim as stated in D2.4 and summarized three experiments of networking scenarios developed in this simulator. Since the last deliverable, the RINASim matured in the tool that can be used for

- getting a deep understanding of RINA mechanisms,
- researching RINA policies and evaluating them in the simulator, and
- analysing various application scenarios in RINA environment by simulating them using RINASim.

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. In this report, several such extensions are described, namely:

- congestion avoidance and control - legacy RED policy is compared to ACC policy,
- scheduling - delay loss and enhanced delay loss scheduling policies are implemented as simulation models and their performance is evaluated,
- routing and forwarding - simulation models for existing distance vector and link-state routing were developed together with TSimple versions and Domain routing policy.

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 next activities related to RINASim represent mainly bug fixing, and extending it with policies that represent additional features. RINASim contributes to PRISTINE project by offering a suitable environment for evaluating fresh research ideas quickly.

PRISTINE's Description of Work (DoW) document contains indicator to measure the utilization of RINASim among partners in PRISTINE project. This metrics is stated in following table.

Table 1. PRISTINE evaluation metric regarding RINASim, as presented in the "Description of Work"

No	Metric	Description
4	Number of simulations done with the RINA simulator developed by PRISTINE	This indicator will measure the relevance and usefulness of the RINA simulator, by keeping track of the usage of the simulator by the consortium partners. It is expected that all the tasks within WP3-4 will use the simulator and document the results achieved with it.

The indicative values for evaluation metric of RINASim are presented in the following table. This table enumerates all completed simulation scenarios.

Table 2. RINASim scenarios

Name, description	Research area	Notes
Set of basic demonstration scenarios	RINA basic principles	The set of demonstration scenarios was made for showing basic principles in RINA, such as, flow handling, resource allocation, traffic relaying, RIB management. These examples are bundled with RINASim.
An advanced demo	RINA principles	The demo presents application communication within a network consisting of all different node types. There are two border routers and a interior router and totally six DIFs of three different ranks. The simulation present variety of RINA mechanims involved in the end-to-end communication.
Aggregated Congestion Control	Congestion control	This demonstration simulates a scenario in which multiple flows were sharing the same bottleneck

Name, description	Research area	Notes
		link in the network. The aim is to analyze how flows can be aggregated and controlled using one congestion controller to reduce the negative effect of competing flows for a shared bandwidth on each other.
Routing	Routing and forwarding	This demonstration aims at analysis of RINA's Forwarding, PDUGE and Routing policies.
Scalable Forwarding with RINA	Routing and forwarding	This demonstration deals with advanced RINA's Forwarding policies. The goal is to evaluate the proposed algorithms for scaling up PDU forwarding.

In addition to listed demonstration scenarios, RINASim is being used as a tool for evaluating newly proposed policies that outcome from research activities. These activities are part of WP3, WP4 and WP6. The list of simulation scenarios is not definitive and the growing is expected. This list will be update at the end of the project.

References

- [rina-intro] J. Day, "An introduction to the Recursive InterNetwork Architecture," January 2015. Available: online[57]
- [networking-is-ipc] J. Day, I. Matta and K. Mattar, "Networking is IPC: a guiding principle to a better internet," in CoNEXT '08 Proceedings of the 2008 ACM CoNEXT Conference , New York, NY, USA, 2008.
- [delta-t-spec] R. Watson, "Delta-t Protocol Specification," Lawrence Livermore Laboratory, December 1981. Available: online[58]
- [delta-t-features] R. Watson, "The Delta-t transport protocol: features and experience," in Proceedings 14th Conference on Local Computer Networks, Minneapolis, USA, 1989.
- [RINA-layer-discovery] E. Trouva, E. Grasa, J. Day and S. Bunch, "Layer discovery in RINA networks," in IEEE 17th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), Barcelona, Spain, 2012.
- [IRM-spec] J. Day, "D-Base-2011-017: IPC Resource Manager (IRM) Specification," Pouzin Society, 2012.
- [Enroll-spec] J. Day, "D-Base-2012-014: Basic Enrollment Specification," Pouzin Society, 2012.
- [Delim1] J. Day, "D-Base-2010-007: Delimiting Module," Pouzin Society, 2009.
- [Delim2] J. Day, "DelimitingGeneral130904: Delimiting Module," Pouzin Society, 2013.
- [EFCP-spec] J. Day, M. Marek, L. Bergesio and M. Tarzan, "EFCPSpec140824_MT_LBJD_MM_v6.6: Error and Flow Control Protocol Specification, Data Transfer + Data Transfer Control," Pouzin Society, 2015.
- [RMT-spec] J. Day, "D-Base-2012-010: Relaying and Multiplexing Task Specification," Pouzin Society, 2012.
- [FA-spec] J. Day, "D-Base-2011-015: Flow Allocator Specification," Pouzin Society, 2011.
- [RA-notes] J. Day, "RINA-RFC-2010-002: Notes on the Resource Allocator," Pouzin Society, 2010.
- [mobj-spec] E. Grasa, S. Bunch and P. deWolf, "Specification of Managed Objects for the Demo DIF," Pouzin Society, 2012.
- [RIB-notes] J. Day, "Notes on the OIB/RIB Daemon," Pouzin Society, 2010.

[57] http://ict-pristine.eu/wp-content/uploads/2014/12/GhentIntroRINAPt1-150119.pdf
[58] http://www.osti.gov/scitech/servlets/purl/5542785

- [isoiec-15953] ISO, "Information technology – Open Systems Interconnection – Service definition for the Application Service Object Association Control Service Element". Patent ISO/IEC 15953:1999, 1999.
- [isoiec-10035-1] ISO, "Information technology – Open Systems Interconnection – Connectionless protocol for the Association Control Service Element: Protocol specification". Patent ISO/IEC 10035-1:1995, 1995.
- [isoiec-9596-1] ISO, "Information technology – Open Systems Interconnection – Common Management Information Protocol: Specification". Patent ISO/IEC 9596-1:1998, 1997.
- [CDAP] S. Bunch, "D-Base-2010-009: CDAP – Common Distributed Application Protocol," Pouzin Society, 2010.
- [CACEP] S. Bunch, J. Day and E. Trouva, "D-Base-2012-016: Common Application Connection Establishment Phase (CACEP)," Pouzin Society, 2012.
- [omnetpp-dwnld] OpenSim Ltd., OMNeT++ Releases, available online[59]
- [github-kvetak] GitHub, RINA Simulator repository, available online[60]
- [ops-rinasimtickets] OpenSource Projects, RINASim Tickets, available online[61]
- [ops-rinasim] OpenSource Projects, RINASim, available online[62]
- [omnetpp-main] OpenSim Ltd., OMNeT++ Discrete Event Simulator, available online[63]
- [omnetpp-inet] OpenSim Ltd., INET Framework, available online[64]
- [omnetpp-ansa] OpenSim Ltd., ANSA Project, available online[65]
- [omnetpp-mixim] OpenSim Ltd., MIXIM Framework, available online[66]
- [omnetpp-oversim] OpenSim Ltd., Oversim Framework, available online[67]
- [omnetpp-veins] OpenSim Ltd., Veins Framework, available online[68]
- [omnetpp-castalia] OpenSim Ltd., Castalia Framework, available online[69]
- [omnetpp-manual] OpenSim Ltd., Manual, available online[70]

[59] http://www.omnetpp.org/omnetpp/category/30-omnet-releases
[60] https://github.com/kvetak/RINA
[61] https://opensourceprojects.eu/p/pristine/rinasimulator/tickets/
[62] https://opensourceprojects.eu/p/pristine/rinasimulator/rinasim/
[63] http://www.omnetpp.org
[64] http://inet.omnetpp.org/
[65] http://nes.fit.vutbr.cz/ansa
[66] http://mixim.sourceforge.net/
[67] http://www.oversim.org/
[68] http://veins.car2x.org/
[69] http://castalia.research.nicta.com.au/index.php/en/
[70] http://www.omnetpp.org/doc/omnetpp/manual/usman.html

- [omnetpp-ide] OpenSim Ltd., IDE in Nutshell, available online[71]
- [omnetpp-demo] OpenSim Ltd., Eclipse, available online[72]
- [omnetpp-userguide] OpenSim Ltd., User Guide, available online[73]
- [omnetpp-highlight] V.Veselý, RINASim C/C++ code highlighter, available online[74]
- [omnetpp-editbox] P.Metel, EditBox | Eclipse Plugins, Bundles and Products - Eclipse Marketplace, available online[75]
- [omnetpp-stats] OpenSim Ltd., Manual, available online[76]
- [RFC6298] Paxson, Vern, et al. "Computing TCP's retransmission timer." RFC 6298. 2011.
- [RFC5681] M. Allman, V. Paxson, and E. Blanton. "TCP congestion Control." RFC 5681, 2009.
- [RED] S. Floyd, V. Jacobson. Random early detection gateways for congestion avoidance. Networking, IEEE/ACM Transactions on, 1993, 1.4: 397-413.

[71] http://www.omnetpp.org/pmwiki/index.php?n=Main.OmnetppInNutshell
[72] http://www.omnest.com/webdemo/ide/demo.html
[73] https://omnetpp.org/doc/omnetpp/UserGuide.pdf
[74] http://nes.fit.vutbr.cz/ivesely/rinasim-highlight.zip
[75] https://marketplace.eclipse.org/content/editbox
[76] https://omnetpp.org/doc/omnetpp/manual/usman.html#sec195