Deliverable-2.4

RINA Simulator; basic functionality

Deliverable Editor: Vladimir Vesely, FIT-BUT

Publication date:	31-January-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:	Simulator, OMNeT++, RINA, event-based
Synopsis:	This document describes the RINA Simulator for OMNeT ++ a.k.a. RINASim.

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.

List of Contributors

Deliverable Editor: Vladimir Vesely, FIT-BUT

FIT-BUT: Marcel Marek, Tomas Hykel, Vladimir Vesely, Ondrej Lichtner, Ondrej Rysavy

i2CAT: Eduard Grasa

CN: Kewin Rausch

Executive Summary

Simulation often serves for validating and verifying new technologies, which do not have yet implementation. Simulation also finds weak-points and drawbacks during test runs and subsequently allows one to enhance development process based on feedbacks. Hence, the implementation of the Recursive Internet Architecture Simulator (RINASim) is a natural step to support the design and development of the RINA SDK. This document introduces RINASim implemented as a framework for the OMNeT++ discrete event simulator. This framework allows the creation of simulation experiments to study RINA mechanisms and policies as well as possible RINA applications. The document consists of an installation walk-through, a high-level concept introduction, key components description and a demonstration of topologies delineation.

Table of Contents

List of Figures

List of definitions

AP or DAP

Application Process or (Distributed Application Process). The instantiation of a program executing in a processing system intended to accomplish some purpose. An Application Process contains one or more tasks or Application-Entities, as well as functions for managing the resources (processor, storage, and IPC) allocated to this AP.

CACEP

Common Application Connection Establishment Phase. CACEP provides the means to establish an application connection between DAPs, allowing them to agree on all the required schemes and conventions to be able to exchange information, optionally authenticating each other.

CDAP

Common Distributed Application Protocol. CDAP enables distributed applications to deal with communications at an object level, rather than forcing applications to explicitly deal with serialization and input/output operations. CDAP provides the application protocol component of a Distributed Application Facility (DAF) that can be used to construct arbitrary distributed applications, of which the DIF is an example. CDAP provides a straightforward and unifying approach to sharing data over a network without having to create specialized protocols.

CEP-id

Connection-endpoint id. A Data Transfer AE-Instance-Identifier unique within the Data Transfer AE where it is generated. This is combined with the destination's CEP-id and the QoS-id to form the connection-id.

DAF

Distributed Application Facility. A collection of two or more cooperating DAPs in one or more processing systems, which exchange information using IPC and maintain shared state. In some Distributed Applications, all members will be the same, i.e. a homogeneous DAF, or may be different, a heterogeneous DAF.

DFT

Directory Forwarding Table. Sometimes referred to as search rules. Maintains a set of entries that map application naming information to

IPC process addresses. The returned IPC process address is the address of where to look for the requested application. If the returned address is the address of this IPC Process, then the requested application is here; otherwise, the search continues. In other words, either this is the IPC process through which the application process is reachable, or may be the next IPC process in the chain to forward the request. The Directory Forwarding table should always return at least a default IPC process address to continue looking for the application process, even if there are no entries for a particular application process naming information.

DIF

Distributed IPC Facility. A collection of two or more Application Processes cooperating to provide Interprocess Communication (IPC). A DIF is a DAF that does IPC. The DIF provides IPC services to Applications via a set of API primitives that are used to exchange information with the Application's peer.

DTCP

Data Transfer Control Protocol. The optional part of data transfer that provide the loosely-bound mechanisms. Each DTCP instance is paired with a DTP instance to control the flow, based on its policies and the contents of the shared state vector.

DTP

Data Transfer Protocol. The required Data Transfer Protocol consisting of tightly bound mechanisms found in all DIFs, roughly equivalent to IP and UDP. When necessary DTP coordinates through a state vector with an instance of the Data Transfer Control Protocol. There is an instance of DTP for each flow.

DTSV

Data Transfer State Vector. The DTSV (sometimes called the transmission control block) provides shared state information for the flow and is maintained by the DTP and the DTCP.

EFCP

Error and Flow Control Protocol. The data transfer protocol required to maintain an instance of IPC within a DIF. The functions of this protocol ensure reliability, order, and flow control as required. It consists of a separate instances of DTP and optionally DTCP, which coordinate through a state vector.

FA

Flow Allocator. The component of the IPC Process that responds to Allocation Requests from Application Processes.

FAI

Flow Allocator Instance. An instance of a FAI is created for each Allocate Request. The FAI is responsible for 1) finding the address of the IPC-Process with access to the requested destination-application; 2) determining whether the requesting Application Process has access to the requested Application Process, 3) selects the policies to be used on the flow, 4) monitors the flow, and 5) manages the flow for its duration.

PCI

Protocol Control Information. The string of octets in a PDU that is understood by the protocol machine which interprets and processes the octets. These are usually the leading bits and sometimes leading and trailing bits.

PDU

Protocol Data Unit. The string of octets exchanged among the Protocol Machines (PM). PDUs contain two parts: the PCI, which is understood and interpreted by the DIF, and User-Data, that is incomprehensible to this PM and is passed to its user.

RA

Resource Allocator. A component of the DIF that manages resource allocation and monitors the resources in the DIF by sharing information with other DIF IPC Processes and the performance of supporting DIFs.

RIB

Resource Information Base. For the DAF, the RIB is the logical representation of the local repository of the objects. Each member of the DAF maintains a RIB. A Distributed Application may define a RIB to be its local representation of its view of the distributed application. From the point of view of the OS model, this is storage.

RMT

Relaying and Multiplexing Task. This task is an element of the data transfer function of a DIF. Logically, it sits between the EFCP and SDU Protection. RMT performs the real time scheduling of sending PDUs on the appropriate (N-1)-ports of the (N-1)-DIFs available to the RMT.

SDU

Service Data Unit. The unit of data passed across the (N)-DIF interface to be transferred to the destination application process. The integrity of an SDU is maintained by the (N)-DIF. An SDU may be fragmented or combined with other SDUs for sending as one or more PDUs.

List of acronyms

ABI	Application Binary Interface.
ACL	Access Control List.
AE	Application Entity.
AP	Application Process.
API	Application Programming Interface.
ASN.1	Abstract Syntax Notation One.
Auth	Authentication module.
CACE	Common Application Connection Establishment module.
CACEP	Common Application Connection Establishment Phase.
CDAP	Common Distributed Application Protocol. CDAppP> RINASim Common Distributed Application Protocol compound module.
CMIP	Common Management Information Protocol.
CRC	Cyclic Redundancy Code.
DA	DIF Allocator.
DAF	Distributed Application Facility.
DAP	Distributed Application Process.
DNS	Domain Name Server.
DHCP	Dynamic Host Configuration Protocol.
DHT	Distributed Hash Table.
DFT	Directory Forwarding Table.
DIF	Distributed IPC Facility.
DRF	Data Run Flag.
DTAE	Data Transfer Application Entity.
DTCP	Data Transfer Control Protocol.
DTP	Data Transfer Protocol.
DTSV	Data Transfer State Vector.
EFCP	Error and Flow Control Protocol.
FA	Flow Allocator.
FAI	Flow Allocator Instance.
GPB	Google Protocol Buffers.
HTTP	Hyper Text Transfer Protocol.
IDD	Inter-DIF Directory.
IPC	Inter Process Communication.
IRM	IPC Resource Manager.
JSON	Java Script Object Notation.

KRPI	Kernel space RINA Plugins Infrastructure.
MA	Management Agent.
MPL	Maximum Packet(PDU) Lifetime.
MPLS	Multi-Protocol Label Switching.
MTBR	Mean Time Between Failures.
MTTR	Mean Time To Recover.
NM-DMS	Network Management Distributed Management System.
NSM	Name Space Manager.
OO	Object Oriented
OOP	Object Oriented Programming
OOD	Object Oriented Development
PCI	Protocol Control Information.
PDU	Protocol Data Unit.
PDUFwd-Gen	PDU Forwarding Table generator.
PM	Protocol Machine.
OS	Operating System.
QoS	Quality of Service.
RA	Resource Allocator.
RIB	Resource Information Base.
RINA	Recursive InterNetwork Architecture.
RPI	RINA Plugins Infrastructure.
RMT	Relaying and Multiplexing Task.
RTT	Round Trip Time.
SDU	Service Data Unit.
SDK	Software Development Kit.
TCP	Transmission Control Protocol.
TTL	Time to Live.
URPI	User space RINA Plugins Infrastructure
UDP	User Datagram Protocol.
VLAN	Virtual Local Area Network.
WFQ	Weighted Fair Queuing.
XML	eXtensible Markup Language.

1. Introduction

During the last two decades, the Internet has become a major communication medium. Its ongoing expansion leads to deployment of a variety of different technologies, which in order to achieve a required functionality, add significant complexity. New technologies bring solutions to new requirements and problems, thus introducing new mechanisms and policies. Mechanism development and its deployment is an exhaustive process that requires the combination of testing, validation and verification. In order to accelerate the discovery of design flaws, it is advisable to create a proof of concept in, e.g., a simulation environment (before creating a real implementation). OMNeT is a powerful and widely used discrete event simulator, which provides ideal foundations for the RINA Simulator implementation. Currently, the largest and most exhaustive model development in OMNeT is represented by models for IP networks (INET library) and wireless communication (MANET library). These two initiatives to provide general and highly customizable models enabling to simulate current IP-based networking demonstrate the capabilities of a modern simulation environment. Encouraged by this fact, our aim is to implement RINASim as the third large modeling and simulation library for OMNeT, enabling anyone to analyze the properties of RINA by means of intrinsic mechanisms or policies and also to perform simulation experiments with RINA applications. This report describes the current status of RINASim development. Though far from being complete, RINASim currently implements most of the basic mechanisms and can be used for demonstration of RINA behavior, simple application scenarios and the initial simulation work that has to be carried out by PRISTINE researchers in Work Package 3.

This report is structured as follows: Chapter 2 describes how to install and run OMNeT with RINASim. Chapter 3 provides a high-level concept overview of RINA nodes and components. Chapter 4 thoroughly describes all the implementation specifics of low-level components. Chapter 5 presents four embedded RINA simulator scenarios, explains their setup and what behavior could be observed in a simulation. The report is summarized in Chapter 6.

2. Installation and configuration

The section explains how to install, configure and deploy the RINASim environment. RINASim is developed as a stand-alone framework for the OMNeT discrete event simulator. The current version is developed for OMNeT v4.5 from 16th July 2014. Nevertheless, no incompatibility problems of RINASim on OMNeT 4.6 have been encountered. However, its full migration towards newer version and further integration testing is on the roadmap for M14.

2.1. OMNeT Installation

RINASim is developed in OMNeT 4.5 but its source codes are fully backward compatible with OMNeT 4.4. The following subsections contain a cookbook that explain where to download, how to install and run the OMNeT IDE for Windows and Linux platforms. Nevertheless, OMNeT is even ported to more developer exotic environments, e.g., Mac or BSD.

2.1.1. Windows installation

1. Download source codes from the official webpages [omnetpp-dwnld]. Beware that in case of 64-bit platform, the simulator and its libraries are still compiled for a 32-bits architecture.
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`.
8. You cannot benefit from the parallel build feature on a Windows platform. Please turn it off in menu *Project* → item *Properties*, tab *C/C++ Build* → subtab *Behavior*, tick off *Enable parallel build*.

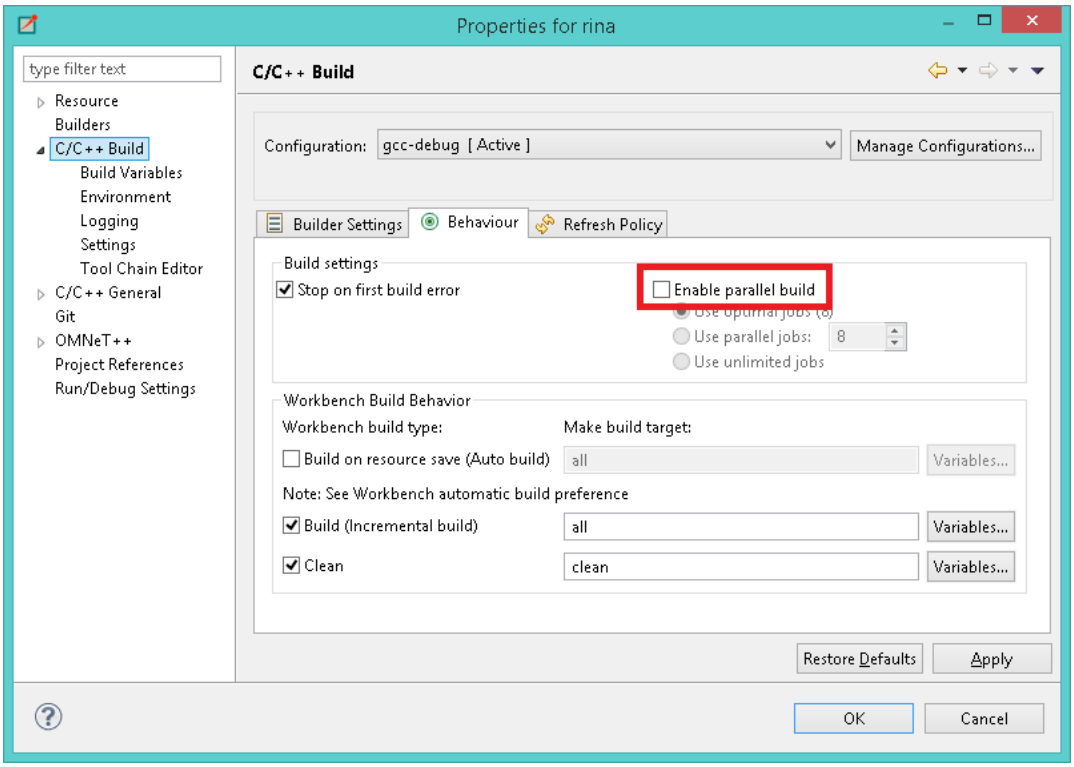

Figure 1. OMNeT IDE Parallel Build

2.1.2. 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-4.5-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.
8. Enjoy the parallel build feature and a native 64-bit environment.

2.2. RINASim Installation

Stable RINASim source codes are periodically published on OpenSourceProjects repository. The reader is encouraged to clone repository localy:

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

FIT-BUT provides support for the newest stable version release. Users can:

1. contact developers via mail (each filename should be accompanied with 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`);

Apart from the stable version release, more ad hoc and thematic source codes are available on the GitHub repository [github-kvetak]. Usually new features are available there sooner than in stable versions. However, no support is provided for those source codes.

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

1. Open the OMNeT IDE and start project import, menu item *File* → *Import...*.
2. Chose *General* and option *Existing Projects into workspace*.
3. Depending on the form of your source codes, chose either *Select root directory* or *Select achive file*.

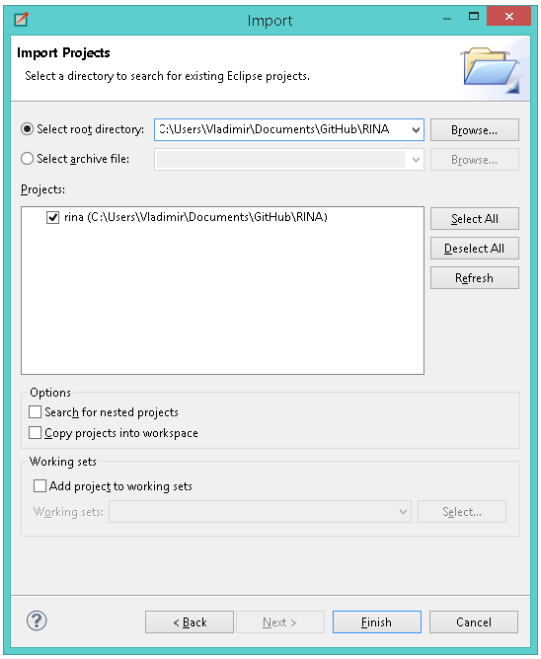

Figure 2. Import Wizard

1. Conclude import via *Finish* button. Now RINASim should be available in the Project explorer under folder `rina`

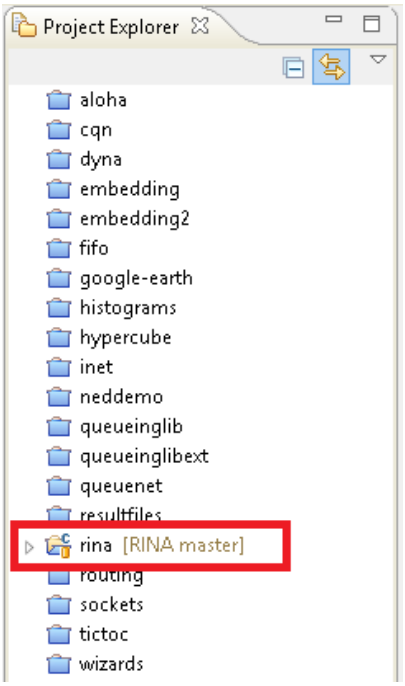

Figure 3. Project Explorer

2.3. 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].

2.3.1. 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.

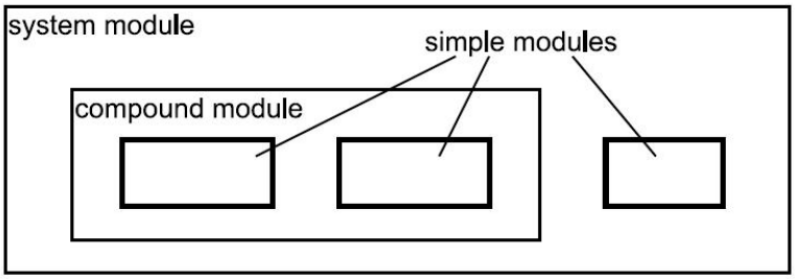

Figure 4. OMNeT module structure

OMNeT is 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 form of PDUs or timer notifications). Messages could be received either from neighbor modules or from the same module (self-messages). A module may contain input (for receiving) and output (for sending) gates. Connections are created between gates. Connection can exist between sibling modules or modules with parent-child relationship.

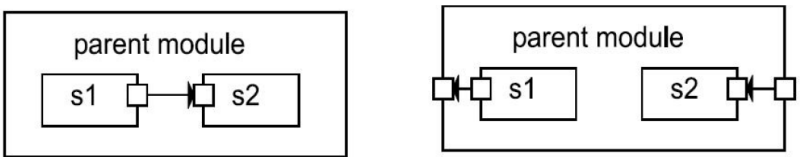

Figure 5. Parent/children modules

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 6. Example of a simple module

Keyword simple defines module´s name TestModule where expected implementation should be in TestModule.cc and TestModule.h. 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 consists of gates definitions (in demo there are two gates, one input gate called in and one output gate called out).

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 7. 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 -→ and bidirectional connections as ←→.

Network modules

The highest level of abstraction is provided by network modules that describe the whole topology of different compound and simple modules. Once again it is described 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 8. Example of a network module

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

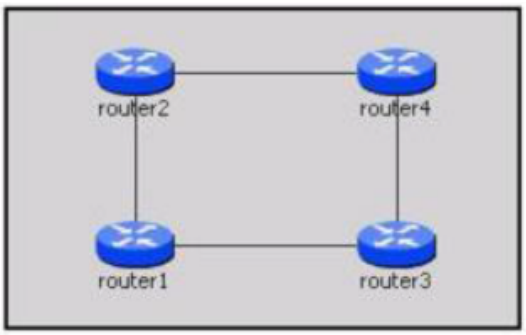

Figure 9. Four routers topology

2.3.2. Simulator and IDE

OMNeT uses the following component architecture:

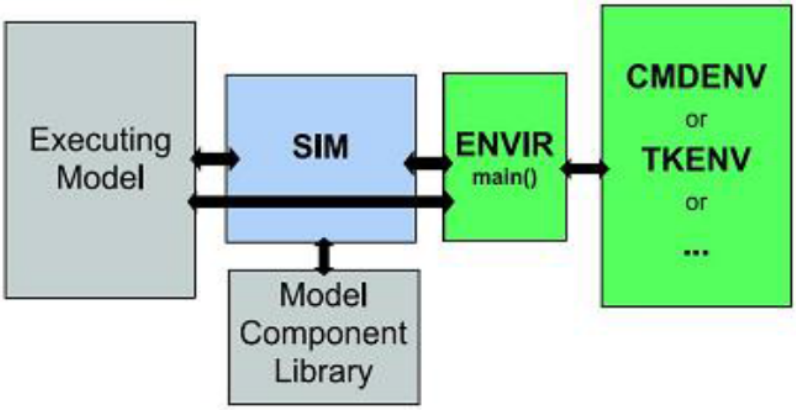

Figure 10. OMNeT component architecture

- 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 ???. 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:

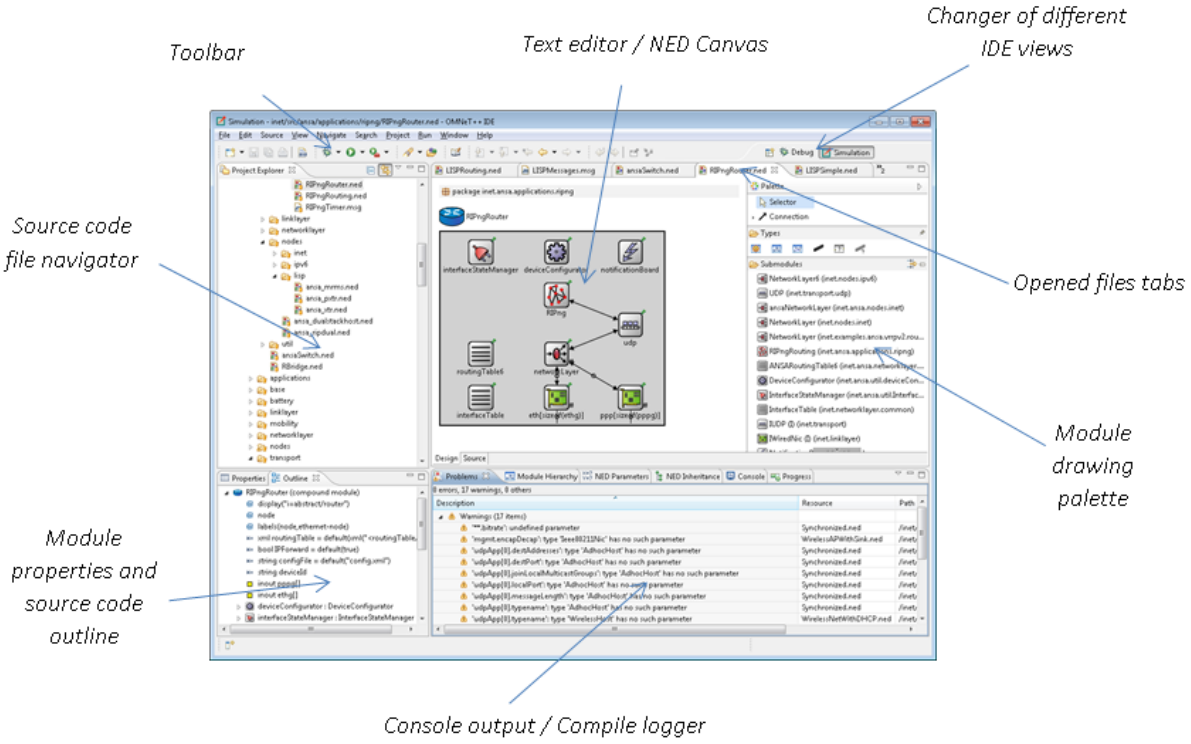

Figure 11. Basic OMNeT++ parts

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

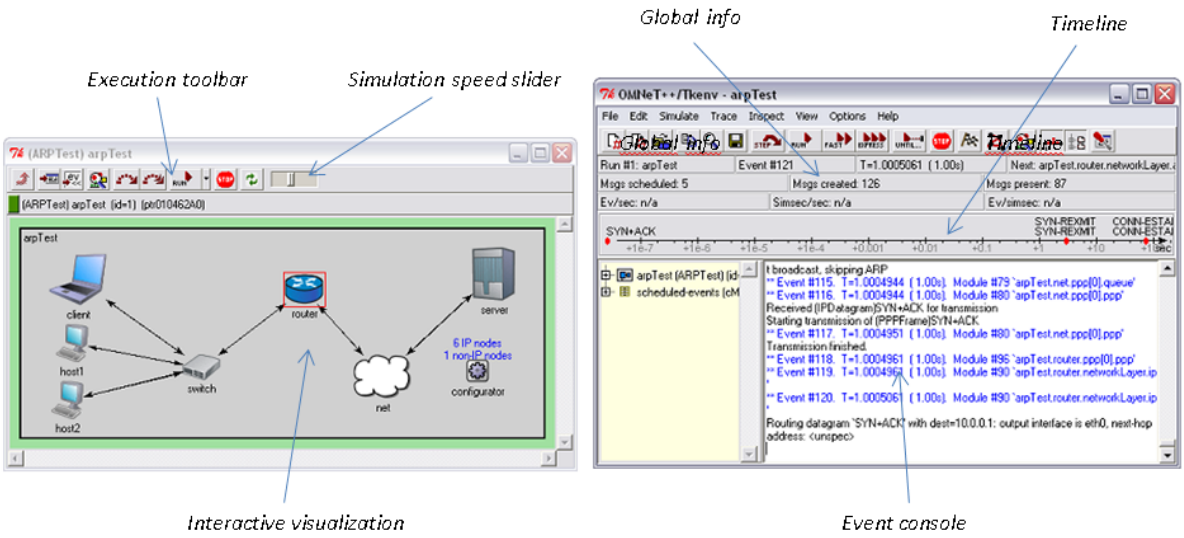

Figure 12. Event logging window

3. High Level Design

This chapter outlines a design of RINA high-level simulation modules, such as the modules modeling the behavior of IPC, data transfer, end or intermediate nodes. The detailed description of submodules of these high-level components is provided in next chapter. 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 identify several places where specifications should be refined to provide more complete and unambiguous information.

3.1. Nodes

RINASim offers a variety of high-level modules simulating the behavior of independent computing system. 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 in addition contains an application process.
- Routers (intermediate nodes) which can be either interior or border. A router is a device that interconnects different underlaying DIFs and often does not run user applications.

In the following subsections we describe the RINASim models that represent host and router devices. These models can be used to quickly set up RINA application simulation experiments. Through parameterization and extension it may be possible to test different deployments and settings without the necessity to implement new host or router models. Full support for extension mechanisms will be released in the next version of RINASim.

3.1.1. Hosts

Host modules represent end-devices that can run Application Processes (AP). 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 the number of APs:

- Host1AP - Host with only a single Application Process. This nodes is suitable for experimenting with internal RINA mechanisms or for learning about RINA mechanisms without incurring additional complexity that stems from simulation of application processes.

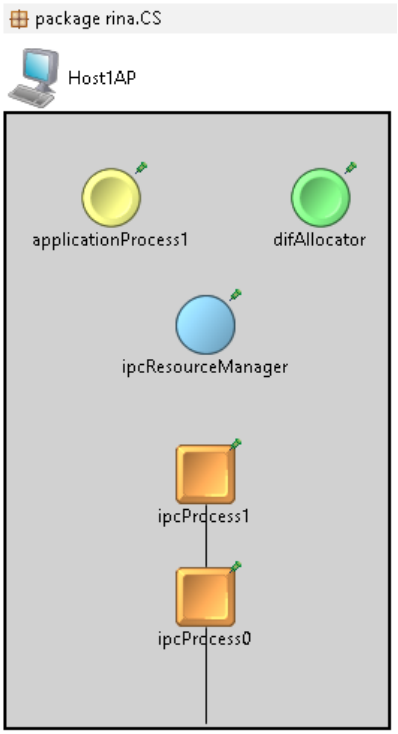

Figure 13. Host1AP

- Host2AP - Host with exactly two Application Processes. This type of node uses two application processes; which is intended for the simulation of scenarios where two application processes may interact in a way that can affect the underlaying RINA stack.

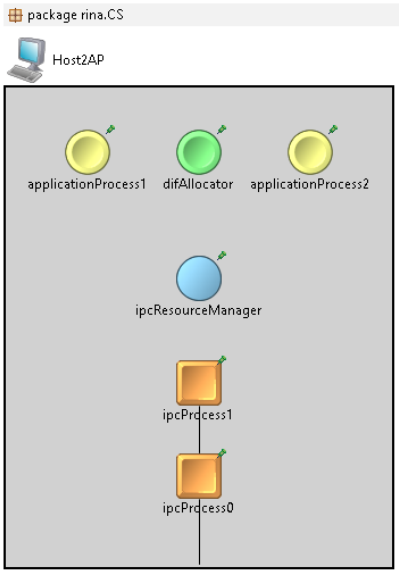

Figure 14. Host2AP

- HostNAP - Host with a configurable number of Application Processes. This host can be useful in more complex scenarios where complex application interaction among multiple processes needs to be analyzed and incorporated in simulation models.

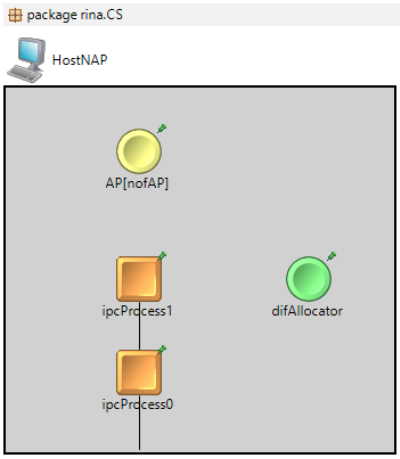

Figure 15. HostNAP

As it can be seen, each host consists of a single DIF Allocator process and a number of IPC processes depending on the depth of RINA stack. In the presented cases, there are two IPC processes. However, it is possible to create nodes with RINA stack of arbitrary depth.

3.1.2. Interior Routers

Interior routers represent devices interconnecting (N-1)-DIFs over a coomon (N)-DIF. Depending on the number of physical interfaces (each

one connected to rank 0-DIF), there are a couple simulation modules available:

- InteriorRouter2Int - Router with a single relay IPC. This IPC operates at 1-DIF layer and represents a bridge over two 0-DIF IPCs. This router represent the simplest possible intermediate device that can perform DIF routing. The simulator also contains models for routers with 3 and 4 interfaces.

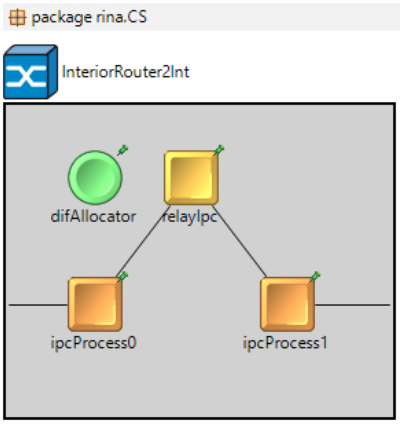

Figure 16. Interior router with 2 interfaces

- InteriorRouterNInt - Router with a single relay IPCP operating over rank 1-DIF and configurable number of 0-DIF IPCs. This is the generic version that enables to configure the number of underlaying interfaces. While it subsumes other variants of interior router models it should be used in scenarios where more than 4 interfaces are necessary. In other case the specialized variants provide a better option as their structure is fixed and easier to work with.

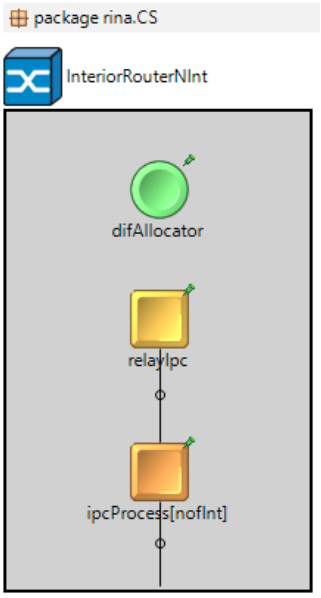

Figure 17. Interior router with N interfaces

3.1.3. Border Routers

Border routers represent devices capable of interconnecting (N-1)-DIFs over mutual (N)-DIF, where some of (N-1)-DIF(s) is/are reachable via (N-2)-DIFs. Currently there is only one Border router model available. A border router with single relay IPC operating over 2-DIF, three 1-DIFs and a single 0-DIF have the following structure:

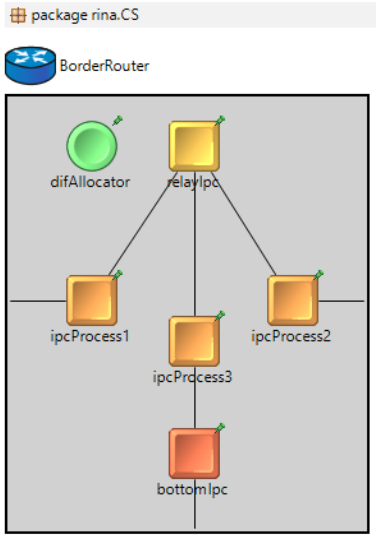

Figure 18. Border router

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 possible combinations but rather to offer the most used ones enabling to quickly

set up simulation scenarios. Certain parameterization can be provided to easier specifying nodes with different configurations, e.g., number of IPC processes at a single layer. As defining new node or router with required structure is not a complicated task the present collection of prepared models seems to be enough.

3.2. DAF Design

Among currently implemented DAF components, there are:

- DIF Allocator,
- Common Distributed Application Protocol (CDAP) Module,
- IPC Resource Manager, and
- Application Process with Application Entities.

Each computing system must have one IPC Resource Manager and one DIF Allocator submodules. There may be one or more Application Processes (AP), where each AP may contain one or more Application Entities.

3.3. DIF Design

Each DIF is represented by an IPC process (IPCP) within the boundaries of a single node regardless whether this node is a host or a router. Each IPCP contains a set of components that are mainly responsible for data transfer and IPC management (including enrollment, allocation, etc.). The following figure shows IPC Process structure, where each subcomponent is described in detail in the following chapter.

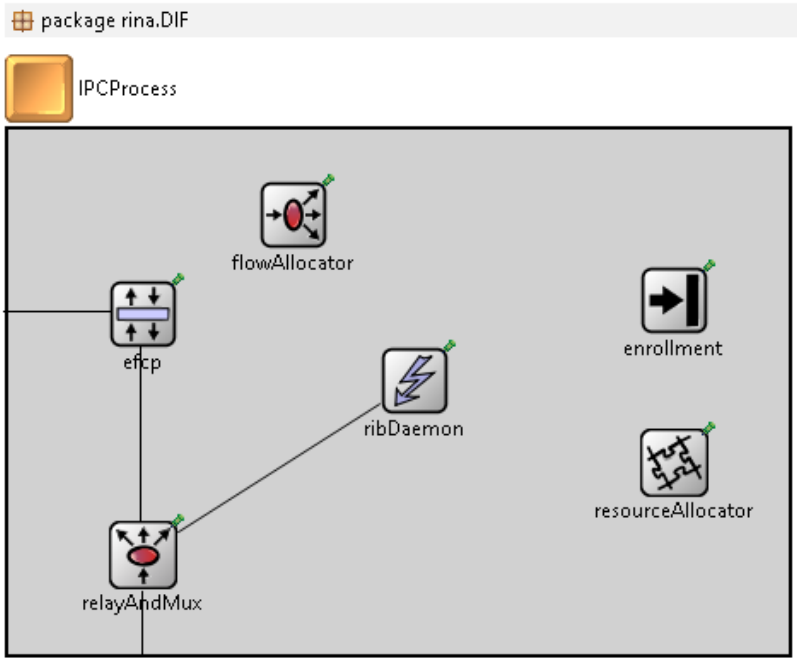

Figure 19. Internal components of the IPC Process

The presented structure includes the following building blocks:

- **Flow Allocator (FA)**, which handles (de)allocation requests from the IPC Resource manager or the RIB daemon. FA itself is structured into the flow table and flow management modules.
- **RIB Deamon (RIBd)**, which receives and sends CDAP messages and notifies other submodules about changes in the Resource Information Base (RIG). This module maintains the state information of the IPCP.
- **Resource Allocator (RA)**, which is the manager of the resources within the IPC Process. It monitors the operation of the IPC Process and makes adjustments to its operation to keep it within the specified operational range.
- **Error and Flow Control Protocol (EFCP)**, which in essence provide the functions of error detection and flow control of data sent and received by the IPCP. The exact function of this module varies with actual policies associated to each flow instance.
- **Relaying and Multiplexing Task (RMT)** is a crossroad for flows within an IPCP. It multiplexes outgoing PDUs from N-EFCP connections to N-1 ports, and demultiplexes incoming PDUs from N-1 ports into N-EFCP connections or relays those PDUs to outgoing N-1 ports if the are directed to other IPC Processes.

3.4. 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. Thus, it is desired to design RINASim in a way that allows for the definition of policies and their easy integration in the simulation models. Rather than providing an exhaustive implementation of policies for each parametrized function, RINASim provides an interface that is used by the core implementation to call functions defined by the selected policy. Users are able to write their own policies and , using a configuration file, plug their policies into the simulation model.

3.4.1. Description

RINASim provides support for user-modifiable policies specifying behavior of miscellaneous parts of RINA stack functionality. An overview of such policies can be found in the documentation of each RINASim component. The separation of mechanism and policy is achieved by splitting the policy procedures into their own separate modules — i.e. each policy invocation is done by calling a method inside the proper policy's module.

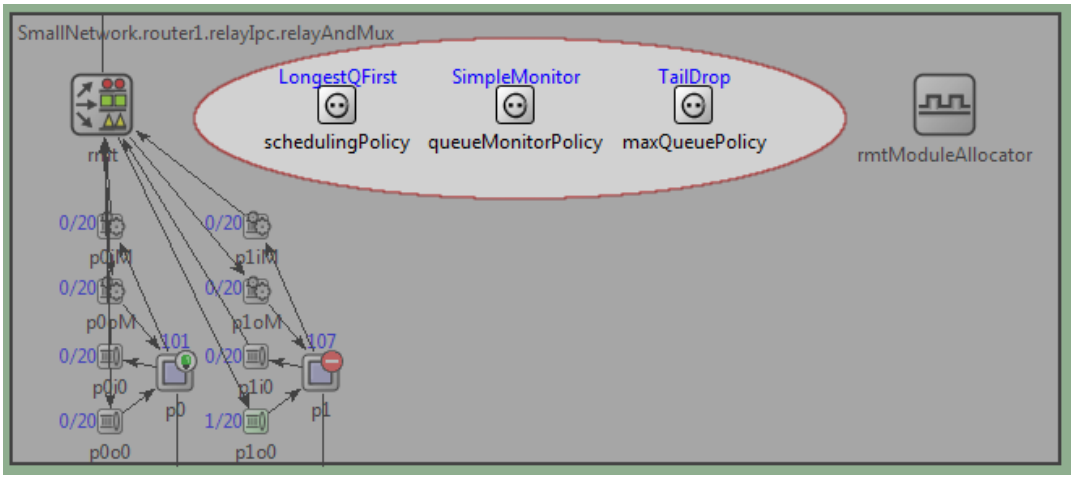

Figure 20. Policy modules

To minimize the need of modifying existing C++/NED source codes, the RINASim policy framework is based on OMNeT NED module interfaces. Instead of placing a simple module with a policy implementation inside the simulation topology, a placeholder interface module is used. The type

of desired policy implementation is then determined at the network setup phase by a parameter placed in an INI config file. This allows for potentially unlimited amount of user policy implementations to be defined and easily switchable via the configuration files.

3.4.2. Using the policy framework

Each policy consists of a NED interface (e.g. "policies/DIF/RA/QueueAlloc/IntQueueAlloc.ned") and a base C++ class (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.

Multiple examples of such polices can be found in "policies/DIF/RMT/" and "policies/DIF/RA".

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.

3.4.3. Example usage

Use case: A user is working with the simulation scenario SimpleRelay[PingFC]. 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). Excerpt from RelayAndMux.ned:

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

Default policies loaded by the simulation:

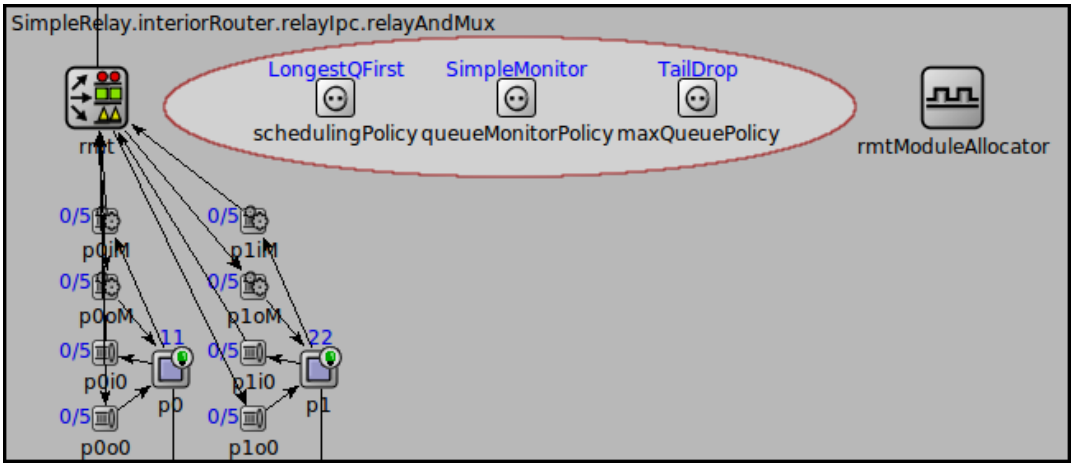

Figure 21. Default policy settings

The user wishes to modify a 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 to prevent congestion. The RED algorithm can be simulated in RINA by two of the RMT policies: QMonitorPolicy (reference implementation "REDMonitor") and MaxQPolicy (reference implementation "REDDropper").

The policy reconfiguration then consist of two steps:

1) making sure the desired implementations are present in their correct policy folders ("src/policies/DIF/RMT/Monitor" and "src/policies/DIF/RMT/MaxQueue"), and

2) Overriding the default policy implementation settings in simulation configuration file omnetpp.ini:

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

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

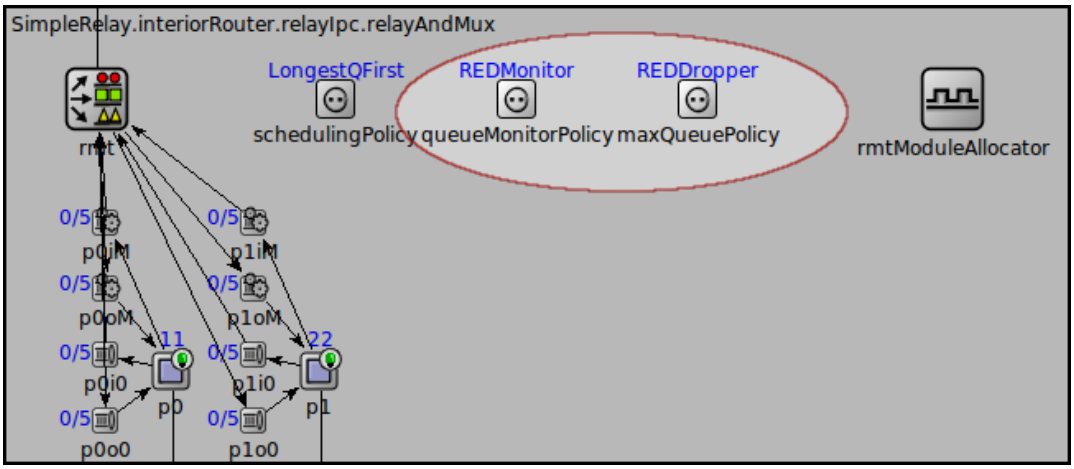

Figure 22. Overriden policy settings

4. Components

This chapter contains the description of the currently implemented and supported components. They are based on the current version of RINA specifications and implemented basic mechanics and policies. They are carefully designed with respect to its extendability and parameterization. It is assumed that for experimenting with RINA concepts these components will be extended with 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.

Each 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

4.1. Application Entity

4.1.1. Image

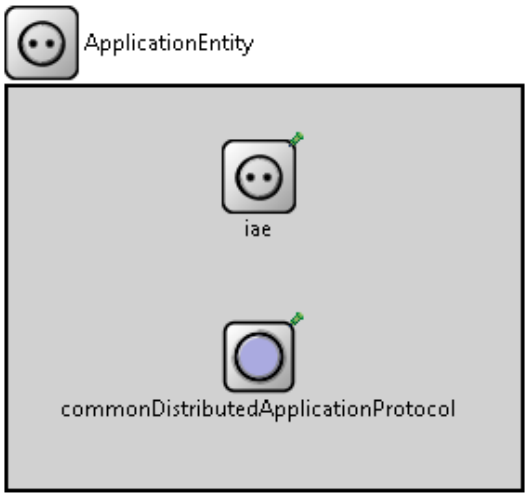

Figure 23. Application Entity

4.1.2. Narrative description

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.

4.1.3. 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".

4.1.4. 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

4.1.5. 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
```

4.1.6. 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_CDAP_DateReceive
SIG_FAI_AllocateRequest
SIG_FAI_DeallocateRequest
SIG_FAI_DeallocateResponse
SIG_FAI_AllocateResponsePositive
```

```
SIG_FAI_AllocateResponseNegative
```

4.1.7. 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.

4.2. Common Distributed Application Protocol

4.2.1. Image

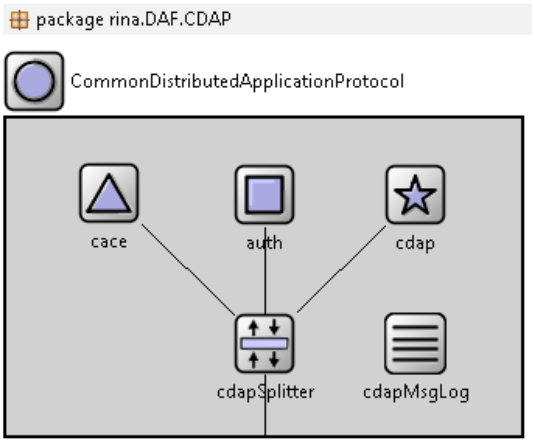

Figure 24. CDAP module

4.2.2. Narrative description

The Common Distributed Application Protocol (CDAP) provides a simple object-based protocol for distributed applications. Currently, it is the part of RIBDaemon and ApplicationEntity compound modules. It prepares CDAP messages to be sent and processes received CDAP messages on behalf of other modules.

4.2.3. Submodules

CDAP is modeled as compound module consisting of five main submodules:

- The **Common Application Connection Establishment (CACE)** module "cace". This module is responsible for the establishment phase of the communication.

- The **Authentication (Auth)** module "auth". This module provides the means for secure authentication of communicating parties during connection initialization.
- The **Common Distributed Application Protocol (CDAP)** module "cdap". This module processes CDAP messages from/to AE.
- **CDAP messages splitter** module "cdapSplitter". The splitter delivers appropriate CDAP message to responsible submodules.
- **CDAP messages logger** module "cdapMsgLog". The logger module is used for debugging and accounting purposes of incoming/outgoing messages.

4.2.4. Source codes

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
"Auth.ned"	Auth simple module
"Auth.h/.cc"	Implementation of Auth core functionality
"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

Filename(s)	Description
"CDAPMessage_m.h/.cc"	C++ implementation of CDAP message classes

4.2.5. NED design

Data-path of interconnected gates for messages:

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

4.2.6. C++ implementation

Registered signals that the CDAP module is emitting:

```
SIG_CDAP_DateReceive
```

Registered signals that the CDAP module is processing:

```
SIG_AE_DataSend
SIG_RIBD_DataSend
```

4.2.7. Side notes

Limitations

1. CACE and Auth are placeholders.
2. CDAP is a stub.

Future work

1. Define interface for CDAP;
2. Implement CACE and Auth module.

4.3. DIF Allocator

4.3.1. Image

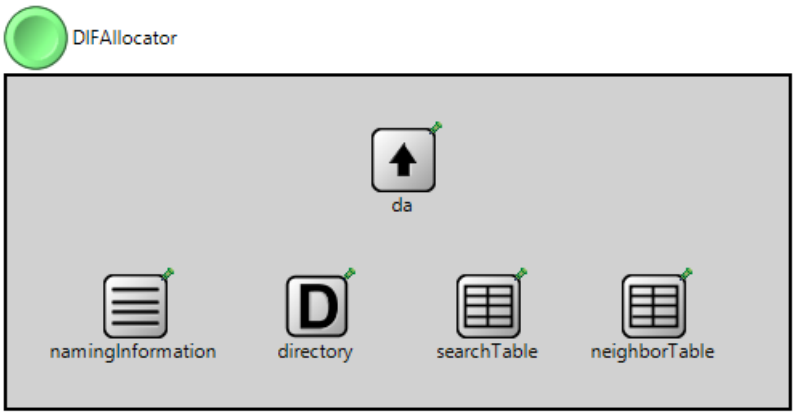

Figure 25. DIF Allocator

4.3.2. Narrative description

The DIF Allocator (DA) component is the successor of component called InterDif Directory (IDD), which is now obsolete in RINA specification. DA is responsible for locating a destination application based on its name. The DA is a component of the DAP's IPC Management that takes Application Naming Information and access control information and returns a list of DIF-names through which the requested application is available.

4.3.3. Submodules

The DA is a compound module containing following five submodules:

- **Naming Information** module "namingInformation" that provides associating synonyms to APNs.
- **Directory** module "directory" that provides a list of supporting DIFs for each AP (defined as a APN-ACL tuple).
- **Search Table** module named "searchTable" that provides mapping between APN and the next DA where to continue the search (DA APN).
- **Neighbor Table** module named "neighborTable" that provides mapping between IDD's peer (IDD APN) and the list of neighboring IDD APNs. This allows RINASim to work as a "oraculum", which knows how the connectivity graph looks like.

- **DIF allocator core** module "da" that implements the DIF allocator logic and provides access interface.

4.3.4. 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

4.3.5. NED design

DA does not have any interconnection between its submodules to send and handle messages.

4.3.6. C++ implementation

DA is currently not receiving/emitting any signals. Usage of DA components is done via direct function calls.

4.3.7. Side notes

Limitations

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

Future work

1. Define interface for DIF allocator;
2. Implement NSM interface with local cache holding DIF allocator responses.

4.4. IPC Resource Manager

4.4.1. Image

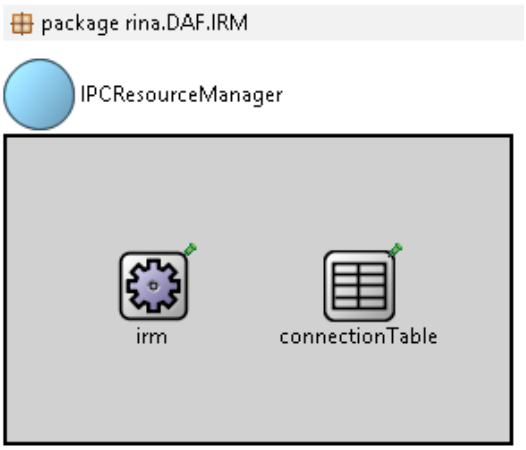

Figure 26. IPC Resource Manager

4.4.2. Narrative description

IPC Resource Manager (IRM) is complex component that is part of each IPC process. It has five main tasks:

1. to query the DIF Allocator in order to localize destination applications,
2. to manage flows with one or more DIFs,
3. to initiate a DAF joining process,
4. to initiate a creation of a new DAF if configured to do so, and

5. to act appropriately when a DIF/DAF is created/lost.

Most notably from the perspective of RINASim, the IRM handles all the application requests imposed on an IPC.

4.4.3. 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
- **Connection Table** - This module maintains the necessary state for IRM correct functionality (the state of the N-1 flows).

4.4.4. 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

4.4.5. NED design

Data-path of interconnected gates for messages from AP to IPC:

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

4.4.6. 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.

4.4.7. Side notes

Future work

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

4.5. Flow Allocator

4.5.1. Image

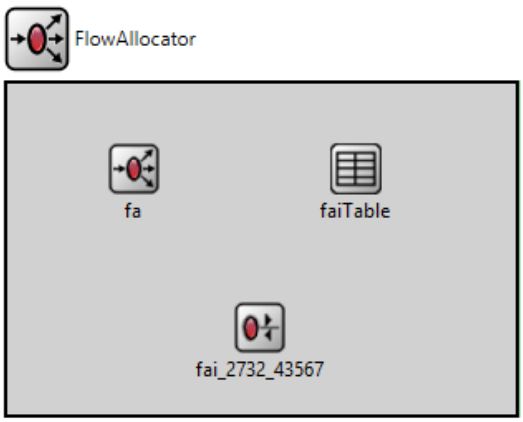

Figure 27. Flow Allocator

4.5.2. Narrative description

The flow Allocator handles flow (de)allocation requests either from the IPC Resource Manager or the RIB Daemon.

4.5.3. Submodules

The Flow Allocator consists of three submodules:

- **Main Flow Allocator** module "fa" acts as the core handler of direct or indirect API calls (through listeners). It instantiates FAIs and delegates program control to them.
- **FA-instance mapping table** module "faiTable", which maintains the necessary state information about which flow is bound to which FAI.
- **FA-instance module** "fai_<PortId>_<CEPId>" which handles the whole flow lifecycle including IRM and EFCP gates (dis)connection.

4.5.4. Source codes

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

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

4.5.5. NED design

FAIs are dynamically created and deleted according to the flow lifecycle.

4.5.6. C++ Implementation

Registered signals that FA module is emitting:

```
SIG_FA_CreateFlowResponseNegative
```

```
SIG_FA_CreateFlowRequestForward
SIG_FA_CreateFlowResponseForward
```

Registered signals that FA module is receiving:

```
SIG_IRM_AllocateRequest
SIG_IRM_DeallocateRequest
SIG_FAI_AllocateResponsePositive
SIG_RIBD_CreateRequestFlow
SIG_RIBD_CreateFlowResponsePositive
```

Registered signals that FAI module is emitting:

```
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 FAI module is receiving:

```
SIG_toFAI_AllocateRequest
SIG_toFAI_AllocateResponseNegative
SIG_AERIBD_AllocateResponsePositive
SIG_RIBD_CreateRequestFlow
SIG_RIBD_CreateFlowResponsePositive
SIG_RIBD_CreateFlowResponseNegative
SIG_RIBD_DeleteRequestFlow
SIG_RIBD_DeleteResponseFlow
```

4.5.7. Side notes

Future work

1. Define interfaces for both FA and FAI;
2. Improve flow lifecycle management (e.g., handling multiple allocation calls).

4.6. Resource Allocator

4.6.1. Image

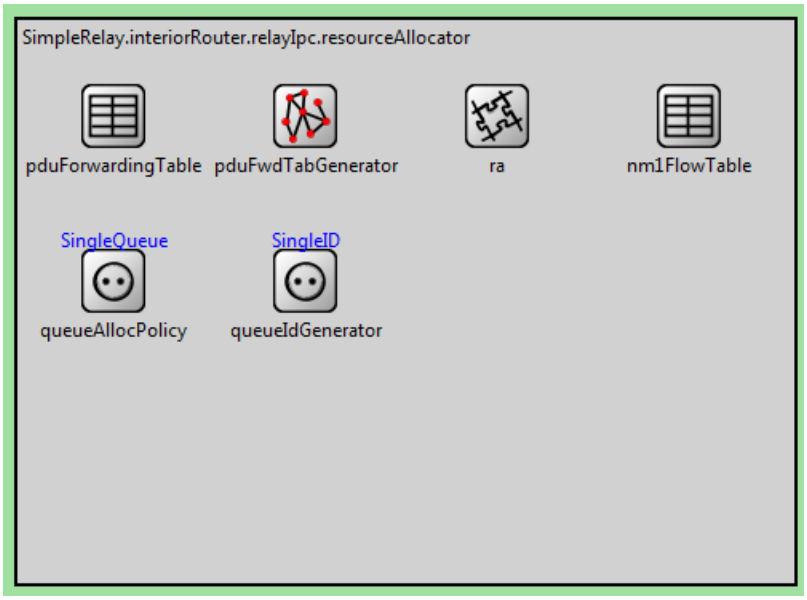

Figure 28. Resource Allocator

4.6.2. Narrative description

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 is customizable by policies. In RinaSim, all the functionality of RA including a policy architecture is encompassed in a single compound module named "resourceAllocator" which is present in every IPC process.

4.6.3. Submodules

The Resource Allocator consists of multiple simple modules of various types, namely:

- ra, the central logic of Resource Allocator that manages connections to other IPC processes via (N-1)-flows as well as the local RMT (i.e. queue allocation and policy adjustments)
- pduForwardingTable, a forwarding table containing the mapping of destination addresses and QoS-ids to output ports that is used by the relaying functionality of the RMT

- pduFwdTabGenerator(abbreviated PDUFTG), a component which, reacting to defined events, uses custom policies to manage pduForwardingTable entries.
- PDUFTGPolicy, the current policy used by pduFwdTabGenerator in order to correctly populate/update the pduForwardingTable.
- nm1FlowTable, a table containing information about the active (N-1)-flows.
- queueAllocPolicy, a policy handling RMT queue allocation.
- queueIdGenerator, a policy generating queue IDs from Flow information and PDUs.

4.6.4. Source codes

Component sources are located in /src/DIF/RA.

Filename(s)	Description
"NM1FlowTable.cc"	implementation of (N-1)-flow table
"NM1FlowTable.ned"	(N-1)-flow table simple module
"NM1FlowTableItem.cc"	implementation (N-1)-flow table entry
"PDUForwardingTable.cc"	implementation of PDU Forwarding Table
"PDUForwardingTable.ned"	PDU Forwarding Table simple module
"PDUForwardingTableEntry.cc"	implementation of PDU Forwarding Table entry
"PDUFTGInfo.cc"	PDUFTG module information of the network state
"PDUFTGListeners.cc"	Listeners for events catched by the PDUFwdTabGenerator module
"PDUFTGUpdate.cc"	PDUFTG update message information
"PDUFwdTabGenerator.cc"	implementation of PDU Forwarding Table Generator
"PDUFwdTabGenerator.ned"	PDU Forwarding Table simple module
"RA.cc"	implementation of RA
"RA.ned"	RA simple module
"RABase.cc"	abstract class for RA implementation
"RAListeners.cc"	signal listeners for RA
"ResourceAllocator.ned"	RA wrapper (compound module)

4.6.5. NED design

ResourceAllocator parameters:

Parameter	Description
"queueAllocPolicyName"	module name of desired QueueAlloc policy
"queueIdGenName"	module name of desired QueueIDGen policy
"pduftgPolicyName"	module name of the desired PDUFTG policy

RA parameters:

Parameter	Description
"qoscubesData"	XML configuration of QoS cubes supported by this IPC process
"flows"	XML configuration of (N-1)-flows to be allocated at the beginning of simulation

4.6.6. Policies

The following policies are currently supported:

Policy folder	Description
"policies/QueueAlloc/"	a folder for QueueAlloc implementations
"policies/QueueIDGen/"	a folder for QueueIDGen implementations
"policies/Forwarding/"	a folder for PDUFTG implementations
"policies/Forwarding/StaticRouting"	implementation of PDUFTG policy for static routing
"policies/Forwarding/DistanceVector"	demo implementation of a Distance Vector forwarding policy

See RINASim policy architecture description[1] for more details.

4.6.7. C++ Implementation

Emitted signals:

"RA-CreateFlowPositive" "RA-CreateFlowNegative"

4.6.8. Side notes

Future work

- Any kind of IPC process performance monitoring is currently nonexistent and shall be implemented when there are clear demands

[1] D24-Policies

- Fine-grained handling of mapping between (N)-QoS and (N-1)-QoS
- Multicast/Broadcast support for PDUFT

4.6.9. PDU Forwarding Table Generator

Image

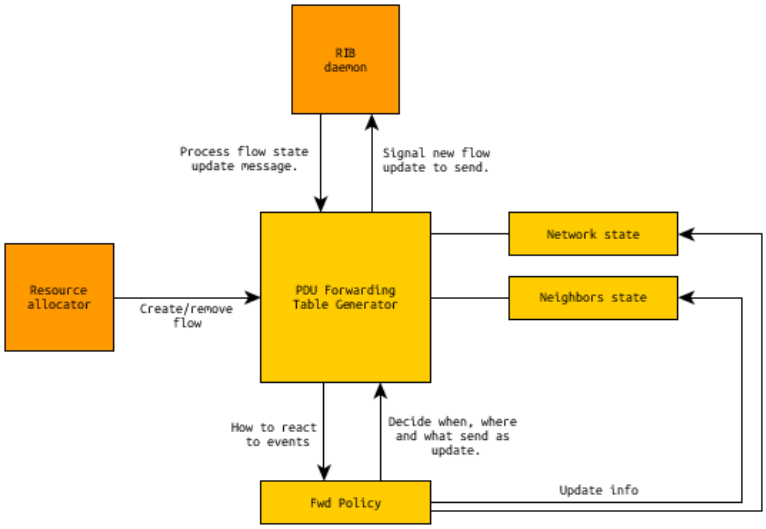

Figure 29. PDU Forwarding Table Generator

Narrative description of functionality

The PDU Forwarding Table Generator (from now on abbreviated as PDUFTG) is a component of the DIF Resource Allocator. The component is in charge of populating the PDU Forwarding Table (PDUFT). The PDUFT is used by the RMT module in order to successfully deliver incoming/outgoing PDUs to the right destination. There are different execution flows which lead to the using of the table entries: whenever traffic from EFCP instances, ports or from the RIB daemon is generated, the RMT looks up the PDUFT in order to resolve the PDU next hop to its destination. What the PDUFT offers is the port to select in order to reach a selected destination with given QoS restrictions. Traffic to the same destination with different QoS requirements will be represented by different entries in the forwarding table.

Policy framework

The PDUFTG comes with a framework which allows developers to implement their own routing policies. The framework reacts at some

important events which occur in the PDUFTG. Such events are: the creation of a flow and the receiving of forwarding update information messages (which identifies when your neighbor decided to exchange information of its vision of the network). The Generator handles only one policy per time. A policy can be assigned at startup, using Omnetpp configuration files, or changed at runtime using the public procedures present in the PDUFTG. When a policy is removed, or unpublished, then the forwarding table is automatically discarded. When a policy is published, then it must perform an initial population of the forwarding table with the Network and Neighbor information present in the PDUFTG (if any).

In order to develop a new routing policy, the developer shall extend the base PDUFTGPolicy class, create the associated Ned module and implement the related functionalities. For compatibility purposes a **Static Routing** policy has already been implemented using the new routing framework. Such policy allows running simulations where the network routing is statically configured.

A simple **Distance Vector** (RIP-like) policy is also included in the simulator. Such policy allows to test the network in a more realistic situation, where the IPCPs in the DIF must exchange information in order to allow communication. Note that this policy has been implemented for educational purposes, in order to provide an example to follow to build new routing logics and as such it is not optimized for performance.

Sub modules list

Network state list

The Network State is a set of information related to the vision of the network from the side of certain IPCP. The network state is populated by the routing policy and describes how IPCPs see the DIF. Such information can be sent later to other IPCPs. It's a matter of policy decide when, what and with whom a certain node IPCP share its network state.

Neighbors state list

The Neighbor State is list of the active neighbors of an IPCP. This set is usually populated by the routing policy when a new flow with some other IPCP has been established. This set is separate from the Network state because it contains the technical information about the port to use

to reach such neighbor. These information are later used to populate the Forwarding table.

PDUFTG policy

The PDUFTG policy is a custom implementation of the decision taken when certain events occurs in the PDUFTG. Policies can deny to react at all to those events, or build sophisticated actions in order to calculate an efficient routing graph. It is all up to the implementation to decide what to do, how to populate network or neighbors information and when to send such information.

Relevant source code files

File path	Description
Src/policies/DIF/RA/Forwarding/ DistanceVector/DistanceVectorPolicy.cc	Distance Vector policy
Src/policies/DIF/RA/Forwarding/ DistanceVector/DistanceVectorPolicy.h	Distance Vector policy
Src/policies/DIF/RA/Forwarding/ DistanceVector/DistanceVectorPolicy.ned	Distance Vector policy
Src/policies/DIF/RA/Forwarding/ DistanceVector/DVPInfo.h	Distance Vector policy message
Src/policies/DIF/RA/Forwarding/ DistanceVector/DVPInfo.cc	Distance Vector policy message
Src/policies/DIF/RA/Forwarding/ StaticRouting/StaticRoutingPolicy.cc	Static Routing policy
Src/policies/DIF/RA/Forwarding/ StaticRouting/StaticRoutingPolicy.h	Static Routing policy
Src/policies/DIF/RA/Forwarding/ StaticRouring/StaticRoutingPolicy.ned`	Static Routing policy
Src/policies/DIF/RA/Forwarding/ PDUFTGPolicy.cc	Generic PDUFTG policy
Src/policies/DIF/RA/Forwarding/ PDUFTGPolicy.h	Generic PDUFTG policy
Src/policies/DIF/RA/Forwarding/ PDUFTGPolicy.ned	Generic PDUFTG policy
Src/DIF/RA/PDUFTGInfo.cc	Network base information
Src/DIF/RA/PDUFTGInfo.h	Network base information
Src/DIF/RA/PDUFTGListeners.cc	Listeners for the PDUFTG module
Src/DIF/RA/PDUFTGListeners.h	Listeners for the PDUFTG module

File path	Description
Src/DIF/RA/PDUFTGNeighbor.cc	PDUFTG view of a neighbor node
Src/DIF/RA/PDUFTGNeighbor.h	PDUFTG view of a neighbor node
Src/DIF/RA/PDUFTGUpdate.cc	PDUFTG update message
Src/DIF/RA/PDUFTGUpdate.h	PDUFTG update message
Src/DIF/RA/PDUFwdTabGenerator.cc	PDU Forwarding Table Generator
Src/DIF/RA/PDUFwdTabGenerator.h	PDU Forwarding Table Generator
Src/DIF/RA/PDUFwdTabGenerator.ned	PDU Forwarding Table Generator

NED design structure

Signals

- **RIBD-ForwardingUpdateReceived**. The PDU Forwarding Table Generator module receive signal from the RIB daemon. Such signals are invoked when an incoming CDAP Write message which contains Network information which shall be elaborated. The PDUFTG will dispatch such information to the currently active policy.
- **PDUFTG-ForwardingInfoUpdate**. The PDU Forwarding Table Generator module invokes a Forwarding Info Update signal when the currently active forwarding policy decide it's time to send such data to a node. The PDUFTG provides a specialized procedure which will take care of the invocation details, and only need an FSUpdateInfo class instance as argument. Such class will contains the necessary information to dispatch the message.

Parameters

- **netStateVisible**. Boolean parameter: this parameter allows to show during the simulation the current situation of the Network state set. Usually these information are shown as a compact table with destination and assigned metric.
- **netStateMod**. String parameter: this parameter allows to select, with a cmodule::getModuleByPath() compatible syntax, the module level where the network state compact report will be seen.
- **netStateAlign**. String parameter: set the alignment of the network state compact report table. Can be set left, right or top the module selected with netStateMod parameter.

Policies

- **StaticRoutingPolicy**. This policy does not perform any adaptive routing. When a new flow is created, this policy just adds an entry to the PDU Forwarding Table using as an input a static network configuration file.
- **DistanceVectorPolicy**. This policy has been implemented for demo purposes. Its job is to relay flow information of each node to its neighbors, incrementing the hop count by one if similar information is not already present in a node network state. If an information with a more performing metric is detected (less hops), then the information is exchanged and the PDU Forwarding table is updated to the new next hop. This is not a complete policy, but can react to the basic topology changes (insertion of new nodes). The policy does not support a crashed link; for the moment it simply does not react or realize that a node goes down.

C++ implementation notes

- **PDUFTG_PRIVATE_DEBUG**. PDUFTG comes with a logging mechanism. Defining this preprocessor symbol at the top the PDUFwdTabGenerator header (the actual version of the simulator has been shipped with such symbol commented), you will find a pduftg.log file in your simulation directory. This file will contains all and only the debugging information produced by PDUFTG module.
- **reportBubbleInfo**. This procedures allow you to produce bubbles information during Oment simulation. These bubbles will be spotted at the same level of netStateMod module variable (see NED Parameters above).

Current limitation and future development plans

Limitations

The notification system of the PDUFTG is limited to flow creation/ destruction and update message received.

Future development

We plan to extend the granularity of the PDUFTG plugin architecture following on the policy creators requests. Depending on the requests the

Generator will be adapted to support other types of events. This will allow the creation of more complex and complete policies in the future.

4.7. RIB Daemon

4.7.1. Image

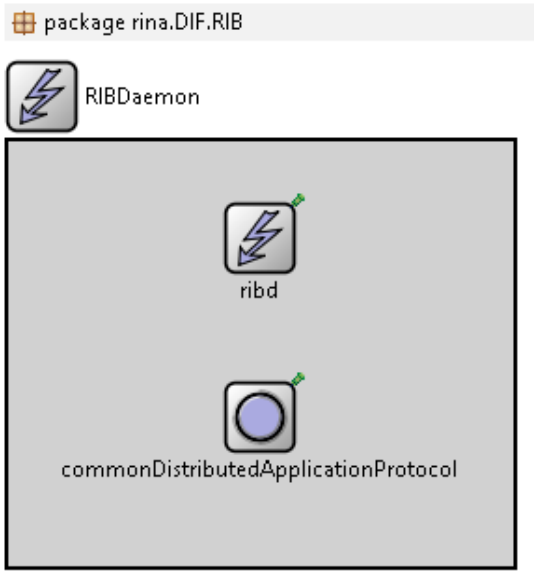

Figure 30. RIB Daemon

4.7.2. Narrative description

The RIBDaemon (RIBd) is the DIF management heart. It receives/ sends CDAP management messages and notifies other submodules about management changes.

4.7.3. Submodules

RIBDaemon consists of two submodules:

- **RIBDaemon** module "ribd" is a core module implementing functionality of RIBDeamon.
- **Common Distributed Application Protocol** module "commonDistributedApplicationProtocol" which implements processing of CDAP messages for "ribd";

4.7.4. Source codes

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

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

4.7.5. NED design

RIBd simulation module design is similar to AE. CDAppP is connected via its southIo gate to RMT.

4.7.6. C++ Implementation

Registered signals that RIBd module is emitting:

```
SIG_RIBD_DataSend
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_ForwardingUpdateReceived
```

Registered signals that RIBd module is 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_CDAP_DateReceive
SIG_FAI_AllocateRequest
SIG_RA_CreateFlowPositive
```

```
SIG_RA_CreateFlowNegative
SIG_PDUFTG_FwdInfoUpdate
```

4.7.7. Side notes

Future work

1. Define RIBd interface;
2. Define CDAP message processing interface.

4.8. Delimiting

4.8.1. Image

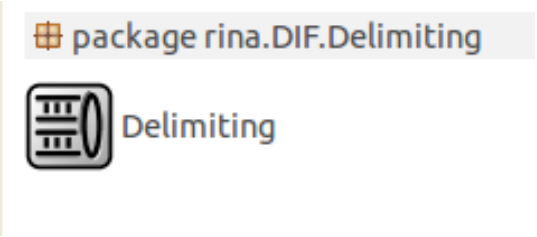

Figure 31. Delimiting

4.8.2. Narrative description

The delimiting Module is responsible for generating payloads for EFCP PDUs from incoming SDUs (by fragmenting or concatenating them) at the sending side; and to recompose the original SDUs at the receiving side. This module is dynamically created as part of the EFCPI compound module. There is usually one Delimiting module for each flow.

4.8.3. Submodules

This module has not submodules.

4.8.4. Source codes

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

Filename(s)	Description
"Delimiting.ned"	Delimiting Module

Filename(s)	Description
"Delimiting.cc"	Delimiting implementation

4.8.5. NED design

Data-path of interconnected gates for messages from FAI to EFCPI:

```
northIo - towards FAI
southIo[] - towards EFCPI
```

4.8.6. C++ Implementation

Delimiting submodule is not sending nor receiving any signals.

4.8.7. Side notes

Limitations

Future work

1. Generate fragments in case SDU.size exceeds MAX_SDU_SIZE on this flow
2. Handle messages from multiple EFCP instances

4.9. Error and Flow Control Protocol

4.9.1. Image

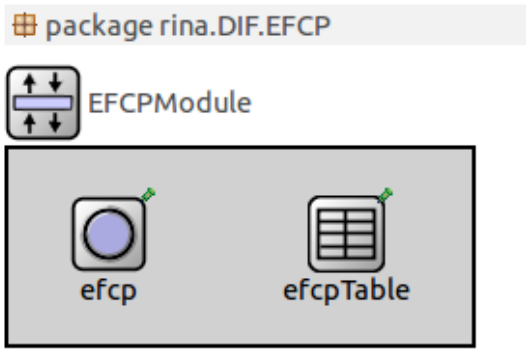

Figure 32. Empty EFCP module without any EFCP instance

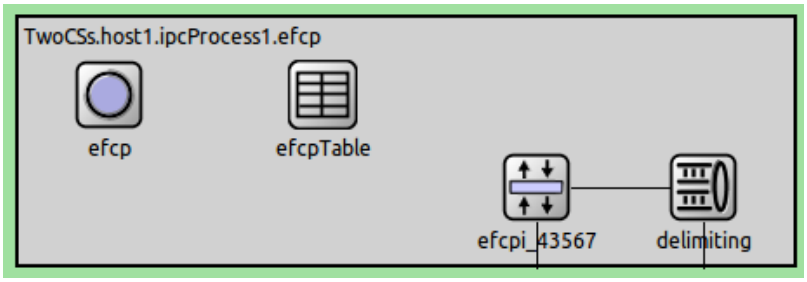

Figure 33. EFCP module with dynamically created Delimiting and EFCP instance modules

4.9.2. Narrative description

The Error and Flow Control Protocol (EFCP) is modeled as one compound module. This module dynamically creates EFCP Instances.Dynamic modules consist of one Delimiting[2] module and (possibly) multiple EFCPI modules per one flow. This EFCPI module itself is a compound module and contains one static module "DTP" and if the flow (QoS requirements) requires *control*, then there is one "DTCP" module.

4.9.3. Submodules

EFCP compound module consists of two static modules:

- **EFCP** module - Creates and deletes EFCPI instances.
- **EFCPTable** module - Holds bindings between Delimiting and EFCPI (DTP and DTCP).
- **Delimiting** module
- **EFCP Instance** module - Implements the EFCP protocol processing logic for a single EFCP connection.

4.9.4. Source codes

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

Filename(s)	Description
"EFCPModule.ned"	EFCP compound module that resides in IPC module
"EFCP.ned"	EFCP module creates and deletes Delimiting and EFCP instancies modules

[2] D24-Rinasim-Delimiting

Filename(s)	Description
"EFCP.cc/h"	EFCP module implementation
"EFCP_defs.h"	EFCP related definitions
"EFCPI.ned"	EFCPI module represents active instance of EFCP
"EFCPTable/EFCPTable.cc/h"	EFCP Table implementation
"EFCPTable/EFCPTableEntry.cc/h"	Entry class for EFCP Table

4.9.5. NED design

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

```
northIo - towards ipc northIo
delimiting.northIo
delimiting.southIo
efcpi_<cep>.northIo
efcpi_<cep>.southIo
southIo - towards RMT
```

4.9.6. C++ Implementation

No registered signals

4.9.7. Side notes

Future work

Implement module layout scheme for meaningful visualization.

4.9.8. EFCP

Image

Figure 34. EFCP module

Narrative description

The EFCP module is responsible for: . Creating a Delimiting instance if it is not already present for this Flow; . Creating EFCPI module; . Creating DTCP module (if necessary); . Create/updating entry in EFCPTable; . Deleting EFCPI and Delimiting on DeallocateFlow request from FA.

Source codes

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

Filename(s)	Description
"EFCP.ned"	EFCP module creates and deletes EFCP instancies
"EFCP.cc/h"	EFCP module implementation

NED design

The EFCP module does not have any gates.

C++ Implementation

The EFCP module does not have any registered signals.

Side notes

Limitations

There are several configurable policies for DTP and DTCP modules. These policies are applied to ALL flows created within this DIF. This way, users can temporarily specify non-default policies without the need to change source code.

Future work

1. Move the policy specification to QoS Cube

4.9.9. EFCPTable

Image

Figure 35. EFCP Table

Narrative description

The EFCPTable stores relations between EFCPI and Delimiting modules.

Source codes

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

Filename(s)	Description
"EFCPTable.ned"	EFCP Table simple module
"EFCPTable.cc/h"	EFCP Table implementation
"EFCPTableEntry.cc/h"	Entry class for EFCP Table

NED design

EFCP Table does not have any gates.

C++ Implementation

EFCP Table does not have any registered signals.

Side notes

EFCP Table is there mainly to support switching between EFCP Instances in case we hit seqNum threshold.

4.9.10. EFCP Instance

Image

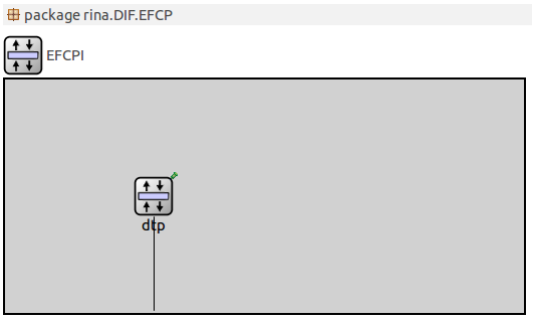

Figure 36. EFCP Instance module at design time

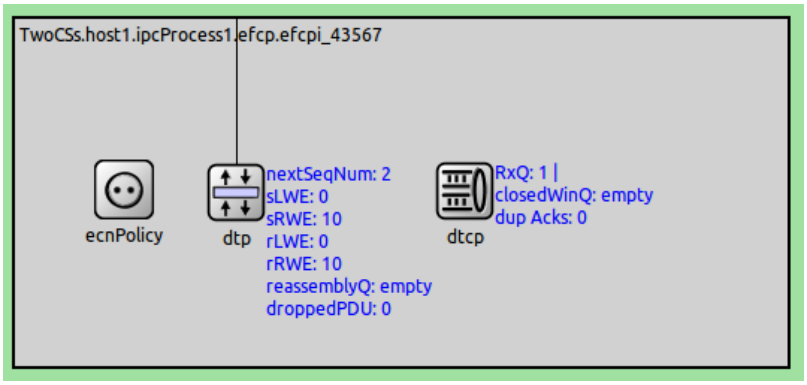

Figure 37. EFCP Instance module runtime, containing a dynamically created policy object and a DTCP object.

Narrative description

Most notably from perspective of RINASim, EFCPI holds together the modules responsible for Data Transfer on a certain flow. It may consist of just a DTP module or DTP, DTCP, DTCPState and a number policies.

Submodules

The EFCP Instance module consists of one static submodule:

- **DTP** - Module provides Data Transfer Protocol.

and several dynamic modules:

- **DTCP** - Provides the *control* part of the data transfer.
- **DTCPState** - Holds state information for DTCP.

- <**name**>**Policy** - If some policy of DTP or DTCP module is specified (ecnPolicy on picture above), this module performs its actions.

Source codes

the compound module itself does not have a implementation, only a NED definition. It consists of following files:

Filename(s)	Description
"EFCPI.ned"	EFCPI module represents active instance of EFCP

NED design

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

```
northIo - towards delimiting
dtp.northIo
dtp.southIo
southIo - towards EFCP Compound Module southIo
```

C++ Implementation

The component does not have any registered signals.

Side notes

Future work

1. Create *empty* transient modules for better positioning connection to south and north gate.

4.9.11. DTP

Image

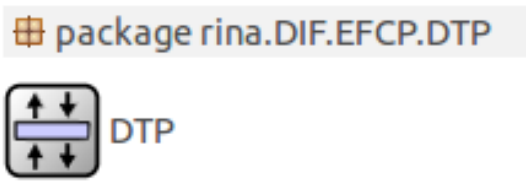

Figure 38. DTP Module

Narrative description

The Data Transfer Protocol accepts user-data fields from the Delimiting module, generates PDUs, and pass them to RMT. If necessary it asks DTCP to perform Retransmission and Flow Control.

Policies

The DTP module is associated with following policies:

- **DTPRcvrInactivityPolicy** - If no PDUs arrive in this time 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).
- **DTPSenderInactivityPolicy** - 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).
- **DTPInitialSeqNumPolicy** - This policy allows some discretion in selecting the initial sequence number, when DRF is going to be sent.

Source codes

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

Filename(s)	Description
"DTP.ned"	DTP module
"DTP.cc/h"	DTP Implementation
"DTPTimers.msg"	DTP related timers definition
"DTPState.cc/h"	Holds state information for DTP (DT-SV)
"DataTransferPDU.msg"	Definition of Data Transfer PDU used to transmit data
"DataTransferPDU.cc/h"	Customized `DataTransferPDU's base class.

NED design

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

```
<efcpi>.northIo
northIo - towards EFCPI's northIo
southIo - towards EFCPI's southIo
```

```
<efcpi>.southIo
```

C++ Implementation

DTP handles incoming SDU from delimiting and produces PDU and send them to RMT. DTP also handles PDUs (Data Transfer and Control) from RMT task. Depending on QoS for this flow delegates DTCP to perform Flow Control and retransmission.

Side notes

The method for receiving PDUs from RMT does not work with maxSeqNumRcvd as it seemed superfluous - need to investigate it more.

Future work

1. Make DTPState standalone module (same as DTCP State).
2. Finish implementation for Allowable gap.
3. Finish implementation for A-Timer.

4.9.12. DTCP

Image

Figure 39. DTCP Module

Narrative description

The Data Transfer Control Protocol (DTCP) handles retransmission and flow control related tasks. From the perspective of RINASim, DTCP is a module that runs policies to update the DTCP state. Policies implement different reactions to situation when error recovery and flow control is expected.

Policies

The DTCP module is associated with following policies:

- **DTCPECNPolicy** - This policy is invoked upon receiving PDU with DRF set in header.
- **DTCPRcvrFCPolicy** - This policy is invoked when a Transfer PDU is received to give the receiving PM an opportunity to update the flow control allocations.
- **DTCPRcvrAckPolicy** - 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.
- **DTCPReceivingFCPolicy** - 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.
- **DTCPSendingAckPolicy** - 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.
- **DTCPLostControlPDUPolicy** - 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 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.
- **DTCPRcvrControlAckPolicy** - This policy is executed by the receiver of Control Ack PDU. Its purpose is to faster recover from PM inconsistency.
- **DTCPSenderAckPolicy** - 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.
- **DTCPFCOverrunPolicy** - 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.
- **DTCPNoOverridePeakPolicy** - 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 which is no override.

- **DTCPTxControlPolicy** - 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.
- **DTCPNoRateSlowDownPolicy** - This policy is used to momentarily lower the send rate below the rate allowed.
- **DTCPReconcileFCPolicy** - 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.
- **DTCPRateReductionPolicy** - 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.

Source codes

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

Filename(s)	Description
`DTCP.ned`	DTCP module
`DTCP.cc/h`	DTCP Implementation
`DTCPTimers.msg`	DTCP related timers definition
`ControlPDU.msg`	Definition of Control PDUs used in Flow Control and Retransmission

NED design

The DTCP module does not have any gates.

C++ Implementation

Side notes

Limitations

MPL and RTT are configurable only through change in source.

Future work

1. Integrate all attributes from FlowControl and Retransmission modules into `DTCPState` (DTCP-SV);
2. Make configurable timers (MPL, RTT)
3. Implement RTT policy

4.9.13. DTCP State

Image

Figure 40. DTCP State Module

Narrative description

The DTCP State (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.

Source codes

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

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

NED design

This module does not have any gates.

C++ Implementation

Dynamically created with DTCP module. No registered signals.

Side notes

Future work

1. Integrate all parameters from Flow Control and Retransmission

4.10. Relaying and Multiplexing Task

4.10.1. Image

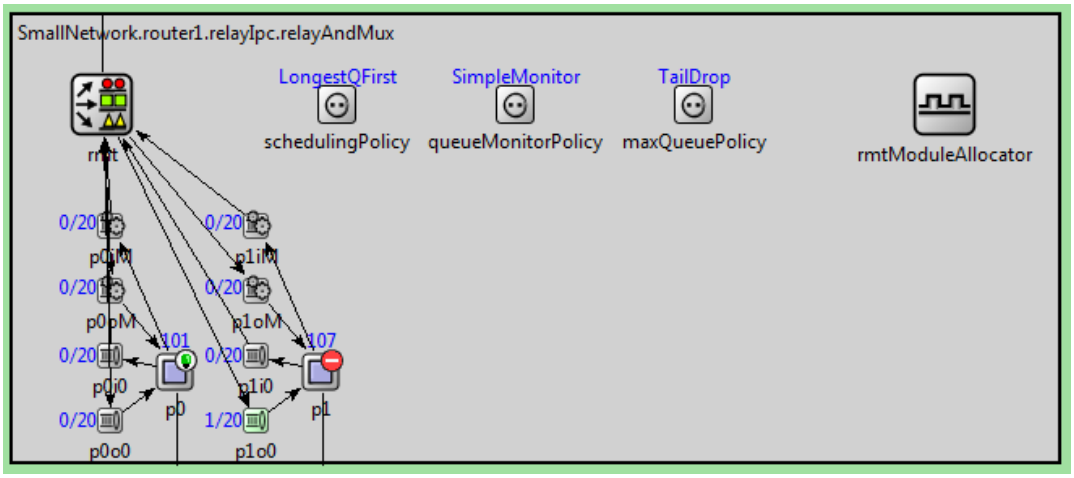

Figure 41. Relaying and Multiplexing Task with three RMT policies

4.10.2. Narrative description

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

- If the address is not an address (or synonym) for this IPC Process, it consults the forwarding table and posts it to the appropriate (N-1)-port-id.
- If the address is one assigned to this IPC Process, the PDU is delivered to either the appropriate EFCP flow or to the RIB Daemon.
- Outgoing PDUs from EFCP-instances or the RIB Daemon are posted to the appropriate (N-1)-port-id.

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.

4.10.3. Submodules

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

Static modules:

- rmt, the central logic of Relaying And Multiplexing task that decides what should be done with messages passing through the module.
- rmtModuleAllocator, a control unit for dynamic modules that provides an API for adding, deleting and reconfiguring RMT queues and ports.
- schedulingPolicy, the scheduler policy which is invoked on events related to servicing of I/O queues.
- queueMonitorPolicy, the monitor policy which is invoked on events related to queue monitoring.
- maxQueuePolicy, the policy used for deciding what to do when queue lengths are overflowing their threshold lengths.

Dynamic modules:

- RMTPort, a representation of one endpoint of an (N-1)-flow.
- RMTQueue, a representation of either input or output queue (the number of RMTQueues per (N-1)-port is a matter of Resource Allocator policies).

4.10.4. Source codes

Component sources are located in /src/DIF/RMT.

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

Filename(s)	Description
"RMTPort.ned"	RMTPort simple module
"RMTQueue.{cc,h}"	implementation of RMTQueue
"RMTQueue.ned"	RMTQueue simple module

4.10.5. 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
"TxQueuingTime"	simulated transmit time for output queues
"RxQueuingTime"	simulated transmit time for input queues
"defaultMaxQLength"	default maximum queue size
"defaultThreshQLength"	default threshold queue size

4.10.6. Policies

Policy folder	Description
"policies/DIF/RMT/Monitor/"	a folder for RMTQMonitorPolicy implementations
"policies/DIF/RMT/Maxqueue/"	a folder for MaxQPolicy implementations
"policies/DIF/RMT/Scheduler/"	a folder for RMTQMonitorPolicy implementations

4.10.7. C++ Implementation

Emitted signals:

- "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)

4.10.8. Side notes

Future work

- Get rid of management-only port queues (currently in use only because CDAP messages are piggy-backed on data flows)
- Separation of mechanism and policy for forwarding decisions
- Cooperation with (N-1)-EFCP on pushback

5. Demonstration Scenarios

The following subsections describe several illustrative scenarios. Each description is accompanied with a list of scheduled events that provide additional information on what can be observed during the simulation run. An interested reader may try them in order to learn more not just about RINASim but also about RINA itself. It is possible to change the parameters or employ different policies to test the various scenarios.

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

1. Brief motivation could be observed in scenario
2. Picture of the scenario
3. List of high-level components employed
4. Initial simulation settings
5. Static XML configuration used to initialize RINA environment
6. Description of the events that may of interest for user

These examples are bundled with RINASim and thus they are easily accessible just by downloading the RINASim package and opening it in the OMNeT++ environment.

5.1. Two Hosts Example

5.1.1. Motivation

This scenario introduces the mechanics of flow allocation and deallocation in RINA on the simplest possible connectivity graph with two directly connected end-hosts.

5.1.2. Scenario

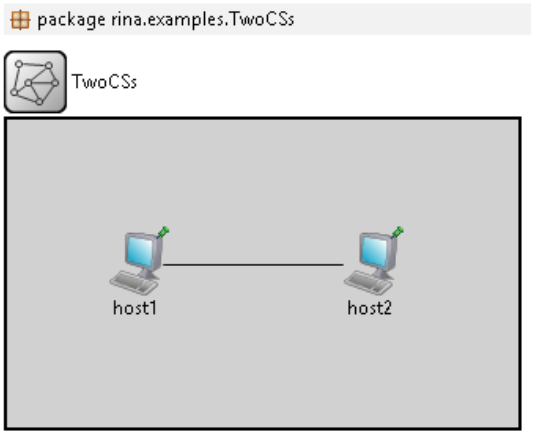

Figure 42. Two directly connected computing systems

5.1.3. High-level components

2× `Host1AP.ned`

5.1.4. Simulation settings in omnetpp.ini

Settings contain the following setup of parameters:

- Used `AE` type is `AEPing`
- APs have assigned APN
- IPCs are assigned an address and DIF name thus creating a unique IPC APN
- DIF allocators are bound with static configuration of mappings
- Two ping scenarios exist. In each one, AP with `AEPing` on Host1 is communicating with Host2's `AEPing` AP

```
[General]
network = TwoCSs
debug-on-errors = true

**.host1.applicationProcess1.apName = "App1"
**.host2.applicationProcess1.apName = "App2"
**.iae.aeName = "Ping"
**.applicationEntity.aeType = "AEPing"
#Static addressing
**.host1.ipcProcess0.ipcAddress = "1"
**.host1.ipcProcess0.difName = "Layer0"
**.host1.ipcProcess1.ipcAddress = "11"
```

```
**.host1.ipcProcess1.difName = "Layer1"

**.host2.ipcProcess0.ipcAddress = "2"
**.host2.ipcProcess0.difName = "Layer0"
**.host2.ipcProcess1.ipcAddress = "22"
**.host2.ipcProcess1.difName = "Layer1"

#DIF Allocator settings
**.host1.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host1']/DA")
**.host2.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host2']/DA")

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

[Config Ping]
#PingApp setup
**.host1.applicationProcess1.applicationEntity.iae.dstApName = "App2"
**.host1.applicationProcess1.applicationEntity.iae.startAt = 10s
**.host1.applicationProcess1.applicationEntity.iae.pingAt =  15s
**.host1.applicationProcess1.applicationEntity.iae.rate = 5
**.host1.applicationProcess1.applicationEntity.iae.stopAt = 20s

[Config Ping-AppQos]
**.host1.applicationProcess1.applicationEntity.iae.dstApName = "App2"
**.host1.applicationProcess1.applicationEntity.iae.startAt = 10s
**.host1.applicationProcess1.applicationEntity.iae.pingAt =  15s
**.host1.applicationProcess1.applicationEntity.iae.rate = 5
**.host1.applicationProcess1.applicationEntity.iae.stopAt = 20s
**.applicationEntity.iae.forceOrder = true
**.applicationEntity.iae.averageBandwidth = 1000000bps
**.applicationEntity.iae.maxAllowGap = 10
**.applicationEntity.iae.delay = 10000 us
```

5.1.5. Static configuration in config.xml

The following configuration introduces:

- DIF Allocator Directory mappings for APs and IPCs
  - AP **App1** is reachable via DIF with name **Layer1** and IPC with address **11**
  - AP **App2** is reachable via DIF with name **Layer1** and IPC with address **22**

  - IPC with address **11** in DIF **Layer1** is reachable via DIF with name **Layer0** and IPC with address **1**
  - IPC with address **22** in DIF **Layer1** is reachable via DIF with name **Layer0** and IPC with address **2**
  - Synonym for **App2** is **AppErr**
- All IPC's resource allocators are setup with two available QoS-cubes

```
<?xml version="1.0"?>
<Configuration>
 <Host id="host1">
  <DA>
   <Directory>
    <APN apn="App1">
     <DIF difName="Layer1" ipcAddress="11" />
    </APN>
    <APN apn="11_Layer1">
     <DIF difName="Layer0" ipcAddress="1" />
    </APN>
    <APN apn="App2">
     <DIF difName="Layer1" ipcAddress="22" />
    </APN>
    <APN apn="22_Layer1">
     <DIF difName="Layer0" ipcAddress="2" />
    </APN>
   </Directory>
   <NamingInfo>
    <APN apn="App2">
     <Synonym apn="AppErr" />
    </APN>
   </NamingInfo>
  </DA>
 </Host>
 <Host id="host2">
  <DA>
   <Directory>
    <APN apn="App1">
     <DIF difName="Layer1" ipcAddress="11" />
    </APN>
    <APN apn="11_Layer1">
     <DIF difName="Layer0" ipcAddress="1" />
    </APN>
    <APN apn="App2">
     <DIF difName="Layer1" ipcAddress="22" />
```

```
    </APN>
    <APN apn="22_Layer1">
     <DIF difName="Layer0" ipcAddress="2" />
    </APN>
   </Directory>
   <NamingInfo>
    <APN apn="App2">
     <Synonym apn="AppErr" />
    </APN>
   </NamingInfo>

  </DA>
</Host>

<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>0</ForceOrder>
  <MaxAllowableGap>10</MaxAllowableGap>
  <Delay>1000000</Delay>
  <Jitter>500000</Jitter>
  <CostTime>0</CostTime>
  <CostBits>0</CostBits>
 </QosCube>
 <QosCube id="2">
  <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>10</MaxAllowableGap>
```

```
    <Delay>1000000</Delay>
    <Jitter>500000</Jitter>
    <CostTime>0</CostTime>
    <CostBits>0</CostBits>
   </QosCube>
  </QoSCubesSet>
</Configuration>
```

5.1.6. Scenario description

Scenario `Ping` has the following phases:

- at t=10s:
  - `Host1.applicationProcess1.ae` initiaties *AllocationRequest*.
  - `Host1.ipcResourceManager.irm` processes *AllocationRequest*. It resolves destination APN to the appropriate IPC Process. Then it forwards *AllocationRequest* towards local IPC Process in the same DIF.
  - `Host1.ipcProcess1.flowAllocator.fa` processes *AllocationRequest*. Because N-1 flow to reach the destination does not exist, it recursively requests the allocation of this flow to the N-1 IPC Process.
  - `Host1.ipcProcess0.flowAllocator.fa` processes flow *AllocationRequest* that should connects it with underlaying N-1 IPC on Host2. In order to do that, it sends signal to RIBd.
  - `Host1.ipcProcess0.ribDaemon.ribd` sends *M_CREATE(flow)* message.
  - `Host2.ipcProcess0.ribDaemon.ribd` receives *M_CREATE(flow)* message and delegates *AllocationRequest* towards `Host2.ipcProcess1.ribd`.
  - `Host2.ipcProcess1.ribDaemon.ribd` accepts allocation and notifies `Host2.ipcProcess0` FA.
  - `Host2.ipcProcess0.flowAllocator.fa` creates application connection between `Host2.ipcProcess1` and `Host2.ipcProcess0` and confirms allocation by triggering M_CREATE_R(flow) on local RIBd.
  - Connection between `Host1.ipcProcess0` and `Host2.ipcProcess0` is successfully established. `Host1.ipcProcess1` may continue with originial flow allocation and sends its own M_CREATE(flow).

- As message passes through `Host1.ipcProcess0`, it is encapsulated into *DataTransferPDU*. It is delivered to `Host2.ipcProcess0`, where is decapsulated and forwarded towards `Host2.ipcProcess1`.
- `Host2.ipcProcess1.ribDaemon.ribd` processes message and local FA notifies destination `Host2.applicationProcess1.ae` about pending allocation.
- `Host2.applicationProcess1.ae` confirms allocation and requests `Host1.ipcResourceManager.irm` to create an application connection between `Host2.applicationProcess1` and `Host2.ipcProcess1`.
- `Host2.applicationProcess1` honors this request and upon successful completion it triggers M_CREATE_R(flow) sending in `Host2.ipcProcess1.ribDaemon.ribd`.
- *M_CREATE_R(flow)* traverses through `Host2.ipcProcess1`, where it is encapsulated into *DataTransferPDU*. It is send to `Host1.ipcProcess0`, where it is decapsulated and delivered to `Host1.ipcProcess1`.
- Upon *M_CREATE_R(flow)* reception, `Host1.ipcProcess1.flowAllocator.fa` notifies `Host1.applicationProcess1.ae` about successful flow allocation.
- `Host1.applicationProcess1.ae` asks `Host1.ipcResourceManager.irm` to finish interconnection. Complete data-path exists between `Host1.applicationProcess1` and `Host2.applicationProcess1`.

- at t=15s:
  - `Host1.applicationProcess1.ae` sends its first of five *M_READ(name)* messages.
  - `Host1.applicationProcess2.ae` responds to it with *M_READ_R(name)* messages.
- at t=20s:
  - `Host1.applicationProcess1.ae` initiaties *DeallocationRequest*.
  - `Host1.ipcResourceManager.irm` processes *DeallocationRequest*. It resolves destination APN to appropriate IPC. Then it forwards *AllocationRequest* towards local IPC in same DIF.

- `Host1.ipcProcess1.flowAllocator.fa` processes *DeallocationRequest* and commands `Host1.ipcProcess1.ribDaemon.ribd` to send *M_DELETE(flow)*.
- *M_CREATE(flow)* passes through data-path until it reaches `Host2.ipcProcess1.ribDaemon.ribd` where it triggers deallocation process in the local FA.
- *DeallocationRequest* is delegated to `Host2.applicationProcess1.ae`, which asks `Host2.ipcResourceManager.irm` to disconnect its portion of the data-path.
- Upon successful completation, `Host2.ipcProcess1.flowAllocator.fa` replies with *M_DELETE_R(flow)*.
- When *M_DELETE(flow)* is delivered to `Host1.ipcProcess1.ribDaemon.ribd`, where it triggers final state of deallocation.
- `Host1.applicationProcess1.ae` is informed about successful deallocation and governs `Host1.ipcResourceManager.irm` to disconnect its portion of the data-path.

5.2. Simple Relay Example

5.2.1. Motivation

This scenario is similar to the previous demo and also shows the basic flow allocation and deallocation mechanics in RINA with a scenario that has been extended to include an interiorRouter between two end-hosts.

5.2.2. Scenario

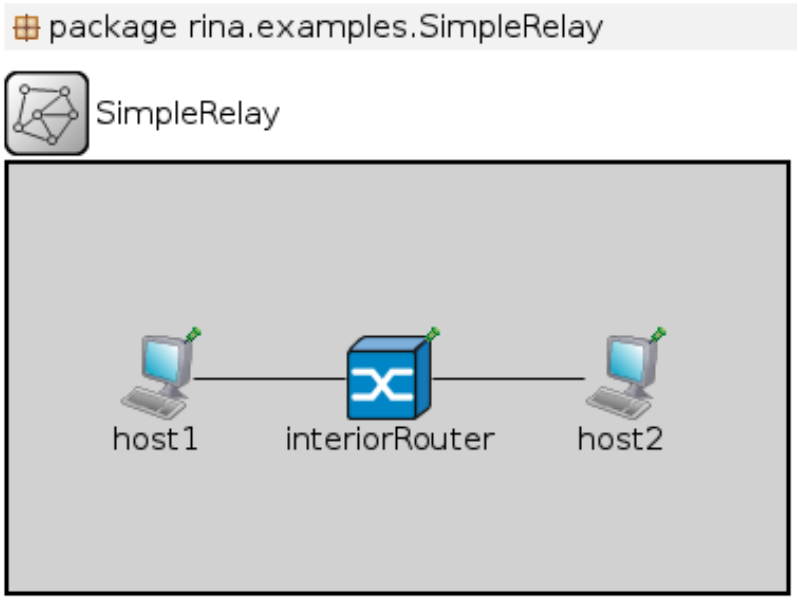

Figure 43. Simple Relay Scenario

5.2.3. High-level components

2× `Host1AP.ned` 1x `InteriorRouter2Int.ned`

5.2.4. Simulation settings in omnetpp.ini

Settings contain following setup of parameters:

- Used `AE` type is `AEPing`
- APs have assigned APN
- IPCPs are assigned an address and DIF name thus creating unique IPC APN
- DIF allocators are bound with static configuration of mappings
- Compared to the TwoCS demo we also have an Interior Router placed between the two hosts
- Four ping scenarios exist. In each one, AP with `AEPing` on Host1 is communicating with Host2's `AEPing` AP
  - Scenario `Ping` is a basic communication example
  - Scenario `PingWithPreallocation` demonstrates preallocation of specified (N-1)-flows on the beginning of simulation (instead of allocating them recursively on the go)

- Scenario `PingWithCongestion` demonstrates one way of handling queues that are overflowing with PDUs
- Scenario `PingWithDiffServ` demonstrates different kinds of RMT queue allocation strategies

```
[General]
network = SimpleRelay
debug-on-errors = true

**.host1.applicationProcess1.apName = "App1"
**.host2.applicationProcess1.apName = "App2"
**.applicationEntity.aeType = "AEPing"
**.iae.aeName = "Ping"

#Static addressing
**.host1.ipcProcess0.ipcAddress = "1"
**.host2.ipcProcess0.ipcAddress = "2"
**.interiorRouter.ipcProcess0.ipcAddress = "3"
**.interiorRouter.ipcProcess1.ipcAddress = "4"

**.host1.ipcProcess1.ipcAddress = "11"
**.host2.ipcProcess1.ipcAddress = "22"
**.interiorRouter.relayIpc.ipcAddress = "33"

**.host1.ipcProcess0.difName = "Layer01"
**.interiorRouter.ipcProcess0.difName = "Layer01"

**.host2.ipcProcess0.difName = "Layer02"
**.interiorRouter.ipcProcess1.difName = "Layer02"

**.host*.ipcProcess1.difName = "Layer11"
**.interiorRouter.relayIpc.difName = "Layer11"

#DIF Allocator settings
**.host1.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host1']/DA")
**.host2.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host2']/DA")
**.interiorRouter.difAllocator.configData = xmldoc("config.xml",
 "Configuration/Router[@id='interiorRouter']/DA")

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

```
[Config Ping]
fingerprint = "9943-e9e1"
#PingApp setup
**.host1.applicationProcess1.applicationEntity.iae.dstApName = "App2"
**.host1.applicationProcess1.applicationEntity.iae.dstAeName = "Ping"
**.host1.applicationProcess1.applicationEntity.iae.startAt = 10s
**.host1.applicationProcess1.applicationEntity.iae.pingAt =  15s
**.host1.applicationProcess1.applicationEntity.iae.rate = 5
**.host1.applicationProcess1.applicationEntity.iae.stopAt = 25s

#Specify AEPing message size
**.host1.applicationProcess1.applicationEntity.iae.size = 256B
#Specify timeout of CreateRequest message
**.fa.createRequestTimeout = 2s

[Config PingWithPreallocation]
fingerprint = "5ce1-13ca"

**.host1.applicationProcess1.applicationEntity.iae.dstApName = "App2"
**.host1.applicationProcess1.applicationEntity.iae.dstAeName = "Ping"
**.host1.applicationProcess1.applicationEntity.iae.startAt = 10s
**.host1.applicationProcess1.applicationEntity.iae.pingAt =  100s
**.host1.applicationProcess1.applicationEntity.iae.rate = 5
**.host1.applicationProcess1.applicationEntity.iae.stopAt = 200s

# flows to allocate at the beginning
**.interiorRouter.relayIpc.resourceAllocator.ra.flows = \
    xmldoc("config.xml", "Configuration/Router[@id='interiorRouter']/
IPC[@id='relayIpc']/RA/Flows")

[Config PingWithCongestion]
fingerprint = "5ce1-13ca"
**.host1.applicationProcess1.applicationEntity.iae.dstApName = "App2"
**.host1.applicationProcess1.applicationEntity.iae.dstAeName = "Ping"
**.host1.applicationProcess1.applicationEntity.iae.startAt = 10s
**.host1.applicationProcess1.applicationEntity.iae.pingAt =  300s
**.host1.applicationProcess1.applicationEntity.iae.rate = 80
**.host1.applicationProcess1.applicationEntity.iae.stopAt = 500s

# make one of the bottom router IPCs become easily congested
**.host1.ipcProcess0.relayAndMux.TxQueuingTime = 50000ms
# use RED as an example congestion control algorithm
**.interiorRouter.ipcProcess1.relayAndMux.qMonitorPolicyName =
 "REDMonitor"
**.interiorRouter.ipcProcess1.relayAndMux.maxQPolicyName = "REDDropper"
```

```
# increase the FA M_CREATE timeout so it doesn't give up too early
**.fa.createRequestTimeout = 100s


[Config PingWithDiffServ]
fingerprint = "5ce1-13ca"

**.host1.applicationProcess1.applicationEntity.iae.dstApName = "App2"
**.host1.applicationProcess1.applicationEntity.iae.dstAeName = "Ping"
**.host1.applicationProcess1.applicationEntity.iae.startAt = 10s
**.host1.applicationProcess1.applicationEntity.iae.pingAt =  100s
**.host1.applicationProcess1.applicationEntity.iae.rate = 5
**.host1.applicationProcess1.applicationEntity.iae.stopAt = 200s

# make all RMTs except the ones in relay IPCs differentiate PDUs by (N)-
flow
**.ipcProcess*.resourceAllocator.queueAllocPolicyName = "QueuePerNFlow"
**.ipcProcess*.resourceAllocator.queueIdGenName = "IDPerNFlow"
# make relay IPCs' RMTs differentiate PDUs by their QoS
**.relayIpc.resourceAllocator.queueAllocPolicyName = "QueuePerNQoS"
**.relayIpc.resourceAllocator.queueIdGenName = "IDPerNQoS"
```

5.2.5. Static configuration in config.xml

The following configuration introduces:

- DIF Allocator Directory mappings for APs and IPCs
  - AP **App1** is reachable via DIF with name **Layer11** and IPC with address **11**
  - AP **App2** is reachable via DIF with name **Layer22** and IPC with address **22**
  - IPC with address **11** in DIF **Layer11** is reachable via DIF with name **Layer01** and IPC with address **1**
  - IPC with address **22** in DIF **Layer11** is reachable via DIF with name **Layer02** and IPC with address **2**
  - IPC with address **33** in DIF **Layer11** is reachable via DIFs **Layer01** and **Layer02** and IPCs with addresses **3** and **4**
  - Neighbour table for both hosts that tells them that they can reach each other through IPC with address **33** in DIF **Layer11**
  - Synonym for **App2** is **AppErr**

- All IPC's resource allocators are setup with two available QoS-cubes

```
<?xml version="1.0"?>
<Configuration>
 <Host id="host1">
  <DA>
   <Directory>
    <APN apn="App1">
     <DIF difName="Layer11" ipcAddress="11" />
    </APN>
    <APN apn="App2">
     <DIF difName="Layer11" ipcAddress="22" />
    </APN>
    <APN apn="11_Layer11">
     <DIF difName="Layer01" ipcAddress="1" />
    </APN>
    <APN apn="22_Layer11">
     <DIF difName="Layer02" ipcAddress="2" />
    </APN>
    <APN apn="33_Layer11">
     <DIF difName="Layer01" ipcAddress="3" />
     <DIF difName="Layer02" ipcAddress="4" />
    </APN>
   </Directory>
   <NamingInfo>
    <APN apn="App2">
     <Synonym apn="AppErr" />
    </APN>
   </NamingInfo>
   <NeighborTable>
    <APN apn="22_Layer11">
     <Neighbor apn="33_Layer11" />
    </APN>
   </NeighborTable>
  </DA>
 </Host>
 <Host id="host2">
  <DA>
   <Directory>
    <APN apn="App1">
     <DIF difName="Layer11" ipcAddress="11" />
    </APN>
    <APN apn="App2">
     <DIF difName="Layer11" ipcAddress="22" />
    </APN>
```

```
    <APN apn="11_Layer11">
     <DIF difName="Layer01" ipcAddress="1" />
    </APN>
    <APN apn="22_Layer11">
     <DIF difName="Layer02" ipcAddress="2" />
    </APN>
    <APN apn="33_Layer11">
     <DIF difName="Layer01" ipcAddress="3" />
     <DIF difName="Layer02" ipcAddress="4" />
    </APN>
   </Directory>
   <NamingInfo>
    <APN apn="App2">
     <Synonym apn="AppErr" />
    </APN>
   </NamingInfo>
   <NeighborTable>
    <APN apn="11_Layer11">
     <Neighbor apn="33_Layer11" />
    </APN>
   </NeighborTable>

  </DA>
 </Host>
 <Router id="interiorRouter">
        <IPC id="relayIpc">
            <RA>
                <Flows>
                    <Flow apn="11_Layer11" qosCube="1"/>
                    <Flow apn="22_Layer11" qosCube="1"/>
                </Flows>
            </RA>
        </IPC>
  <DA>
   <Directory>
    <APN apn="App1">
     <DIF difName="Layer11" ipcAddress="11" />
    </APN>
    <APN apn="App2">
     <DIF difName="Layer11" ipcAddress="22" />
    </APN>
    <APN apn="11_Layer11">
     <DIF difName="Layer01" ipcAddress="1" />
    </APN>
    <APN apn="22_Layer11">
     <DIF difName="Layer02" ipcAddress="2" />
```

```
    </APN>
    <APN apn="33_Layer11">
     <DIF difName="Layer01" ipcAddress="3" />
     <DIF difName="Layer02" ipcAddress="4" />
    </APN>
   </Directory>
  </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>0</ForceOrder>
  <MaxAllowableGap>10</MaxAllowableGap>
  <Delay>1000000</Delay>
  <Jitter>500000</Jitter>
  <CostTime>0</CostTime>
  <CostBits>0</CostBits>
  <ATime>0</ATime>
 </QoSCube>
 <QoSCube id="2">
  <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>10</MaxAllowableGap>
  <Delay>1000000</Delay>
  <Jitter>500000</Jitter>
  <CostTime>0</CostTime>
  <CostBits>0</CostBits>
```

```
    <ATime>0</ATime>
   </QoSCube>
  </QoSCubesSet>
</Configuration>
```

5.2.6. Scenario description

The scenario `Ping` has the following notable phases:

- at t=10s:
  - `Host1.applicationProcess1.ae` initiaties *AllocationRequest*.
  - `Host1.ipcResourceManager.irm` processes *AllocationRequest*. It resolves destination APN to appropriate IPC. Then it forwards *AllocationRequest* towards local IPCP in the same DIF.
  - `Host1.ipcProcess1.flowAllocator.fa` processes *AllocationRequest*. Because N-1 flow to reach destination does not exist, it recursively requests an N-1 flow to the underlaying ICP Process.
  - `Host1.ipcProcess0.flowAllocator.fa` processes flow *AllocationRequest* that should connects it with underlaying N-1 IPC on InteriorRouter. In order to that, it sends signal to RIBd.
  - `Host1.ipcProcess0.ribDaemon.ribd` sends *M_CREATE(flow)* message.
  - `InteriorRouter.ipcProcess0.ribDaemon.ribd` receives *M_CREATE(flow)* message and delegates *AllocationRequest* towards `InteriorRouter.relayIpc.ribd`.
  - `InteriorRouter.relayIpc.ribDaemon.ribd` accepts allocation and notifies `InteriorRouter.ipcProcess0` FA.
  - `InteriorRouter.ipcProcess0.flowAllocator.fa` creates connection between `InteriorRouter.relayIpc` and `InteriorRouter.ipcProcess0` and confirms allocation by triggering M_CREATE_R(flow) on local RIBd.
  - Connection between `Host1.ipcProcess0` and `InteriorRouter.ipcProcess0` is successfully established. `Host1.ipcProcess1` may continue with originial flow allocation and sends its own M_CREATE(flow) directed at Host2.

- When `InteriorRouter.relayIpc` recieves the M_CREATE(flow) message from `Host1.ipcProcess1` it first needs to create a connection to the `Host2.ipcProcess1`.
- This happens the same way as when creating connection from Host1 to InteriorRouter - by first creating a connection between `InteriourRouter.ipcProcess1` and `Host2.ipcProcess0`.
- When this connection is created `InteriorRouter.relayIpc` forwards the M_CREATE(flow) message from `Host1.ipcProcess1` to `Host2.ipcProcess1`.
- `Host2.ipcProcess1.ribDaemon.ribd` processes message and local FA notifies destination `Host2.applicationProcess1.ae` about pending allocation.
- `Host2.applicationProcess1.ae` confirms allocation and bothers `Host2.ipcResourceManager.irm` to create connection between `Host2.applicationProcess1` and `Host2.ipcProcess1`.
- `Host2.applicationProcess1` honors this request and upon successful completation it triggers M_CREATE_R(flow) sending in `Host2.ipcProcess1.ribDaemon.ribd`.
- Upon *M_CREATE_R(flow)* reception, `Host1.ipcProcess1.flowAllocator.fa` notifies `Host1.applicationProcess1.ae` about successful allocation.
- `Host1.applicationProcess1.ae` asks `Host1.ipcResourceManager.irm` to finish interconnection. Complete data-path exists between `Host1.applicationProcess1` and `Host2.applicationProcess1`.

- at t=15s:
  - `Host1.applicationProcess1.ae` sends its first ping request as a *M_READ(name)* message.
  - `Host1.applicationProcess2.ae` responds to it with *M_READ_R(name)* messages.
- at t=25s:
  - `Host1.applicationProcess1.ae` initiaties *DeallocationRequest*.

- `Host1.ipcResourceManager.irm` processes *DeallocationRequest*. It resolves destination APN to appropriate IPC. Then it forwards *DeallocationRequest* towards local IPC in same DIF.
- `Host1.ipcProcess1.flowAllocator.fa` processes *DeallocationRequest* and commands `Host1.ipcProcess1.ribDaemon.ribd` to send *M_DELETE(flow)*.
- *M_CREATE(flow)* passes through data-path until it reaches `Host2.ipcProcess1.ribDaemon.ribd` where it triggers deallocation process in local FA.
- *DeallocationRequest* is delegated to `Host2.applicationProcess1.ae`, which asks `Host2.ipcResourceManager.irm` disconnect its portion of the data-path.
- When *M_DELETE(flow)* is delivered to `Host1.ipcProcess1.ribDaemon.ribd`, where it triggers final state of deallocation.
- `Host1.applicationProcess1.ae` is informed about successful deallocation and governs `Host1.ipcResourceManager.irm` to disconnect its portion of the data-path.

5.3. Small Network Example

5.3.1. Motivation

This scenario introduces multiple interior routers between two communicating hosts.

5.3.2. Scenario

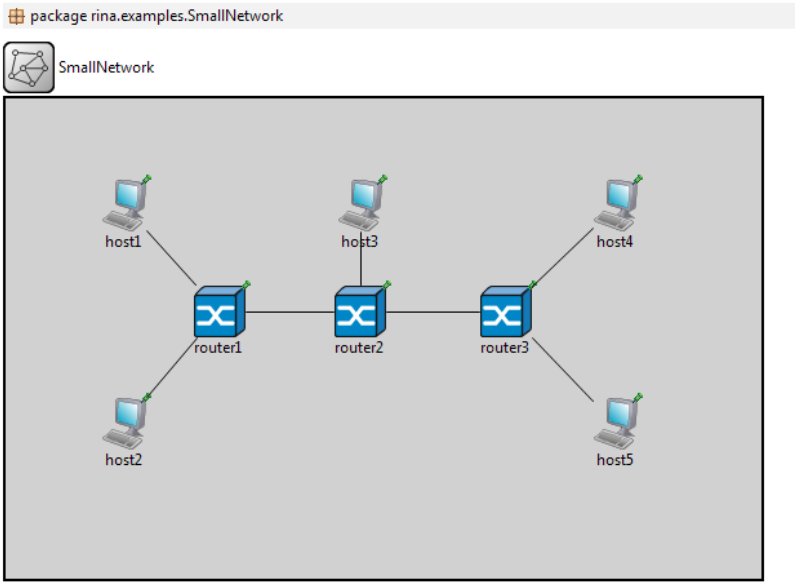

Figure 44. Small network scenario

5.3.3. High-level components

5× `Host1AP.ned`

3x `InteriorRouter3Int.ned`

5.3.4. Simulation settings in omnetpp.ini

Settings contain the following setup of parameters:

- Used `AE` type is `AEPing`
- APs have assigned APN
- DIF allocators are binded with static configuration of mappings
- There's a single scenario present demonstrating communication between host1 and host5

```
[General]
network = SmallNetwork

**.host1.applicationProcess1.apName = "App1"
**.host2.applicationProcess1.apName = "App2"
**.host3.applicationProcess1.apName = "App3"
**.host4.applicationProcess1.apName = "App4"
**.host5.applicationProcess1.apName = "App5"
**.applicationEntity.aeType = "AEPing"
```

```
**.iae.aeName = "Ping"

#Static addressing: lower IPC layer
**.host1.ipcProcess0.ipcAddress = "1"
**.host2.ipcProcess0.ipcAddress = "2"
**.host3.ipcProcess0.ipcAddress = "3"
**.host4.ipcProcess0.ipcAddress = "4"
**.host5.ipcProcess0.ipcAddress = "5"

**.router1.ipcProcess0.ipcAddress = "6"
**.router1.ipcProcess1.ipcAddress = "7"
**.router1.ipcProcess2.ipcAddress = "8"
**.router2.ipcProcess0.ipcAddress = "9"
**.router2.ipcProcess1.ipcAddress = "10"
**.router2.ipcProcess2.ipcAddress = "11"
**.router3.ipcProcess0.ipcAddress = "12"
**.router3.ipcProcess1.ipcAddress = "13"
**.router3.ipcProcess2.ipcAddress = "14"

**.host1.ipcProcess0.difName = "Layer01"
**.router1.ipcProcess0.difName = "Layer01"

**.host2.ipcProcess0.difName = "Layer02"
**.router1.ipcProcess1.difName = "Layer02"

**.router1.ipcProcess2.difName = "Layer03"
**.router2.ipcProcess0.difName = "Layer03"

**.router2.ipcProcess1.difName = "Layer04"
**.router3.ipcProcess0.difName = "Layer04"

**.router2.ipcProcess2.difName = "Layer05"
**.host3.ipcProcess0.difName = "Layer05"

**.router3.ipcProcess1.difName = "Layer06"
**.host4.ipcProcess0.difName = "Layer06"

**.router3.ipcProcess2.difName = "Layer07"
**.host5.ipcProcess0.difName = "Layer07"

#Static addressing: higher IPC layer
**.host1.ipcProcess1.ipcAddress = "101"
**.host2.ipcProcess1.ipcAddress = "102"
**.host3.ipcProcess1.ipcAddress = "103"
**.host4.ipcProcess1.ipcAddress = "104"
**.host5.ipcProcess1.ipcAddress = "105"
```

```
**.router1.relayIpc.ipcAddress = "106"
**.router2.relayIpc.ipcAddress = "107"
**.router3.relayIpc.ipcAddress = "108"

**.host*.ipcProcess1.difName = "Layer11"
**.router*.relayIpc.difName = "Layer11"

#DIF Allocator settings
**.host1.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host1']/DA")
**.host2.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host2']/DA")
**.host3.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host3']/DA")
**.host4.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host4']/DA")
**.host5.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='host5']/DA")

**.router1.difAllocator.configData = xmldoc("config.xml", "Configuration/
Router[@id='router1']/DA")
**.router2.difAllocator.configData = xmldoc("config.xml", "Configuration/
Router[@id='router2']/DA")
**.router3.difAllocator.configData = xmldoc("config.xml", "Configuration/
Router[@id='router3']/DA")

#Directory settings
**.host2.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='host1']/DA")
**.host3.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='host1']/DA")
**.host4.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='host1']/DA")
**.host5.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='host1']/DA")

**.router2.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Router[@id='router1']/DA")
**.router3.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Router[@id='router1']/DA")

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

[Config Ping]
```

```
fingerprint = "bfa8-e8e3"

#PingApp setup
**.host1.applicationProcess1.applicationEntity.iae.dstApName = "App5"
**.host1.applicationProcess1.applicationEntity.iae.dstAeName = "Ping"
**.host1.applicationProcess1.applicationEntity.iae.startAt = 10s
**.host1.applicationProcess1.applicationEntity.iae.pingAt =  100s
**.host1.applicationProcess1.applicationEntity.iae.rate = 5
**.host1.applicationProcess1.applicationEntity.iae.stopAt = 200s
```

5.3.5. Static configuration in config.xml

The following configuration introduces:

- DIF Allocator Directory mappings for APs and IPCs
  - AP **App1** is reachable via DIF with name **Layer11** and IPC with address **101**
  - AP **App5** is reachable via DIF with name **Layer11** and IPC with address **105**
  - IPC with address **101** in DIF **Layer11** is reachable via DIF with name **Layer01** and IPC with address **1**
  - IPC with address **105** in DIF **Layer11** is reachable via DIF with name **Layer07** and IPC with address **5**
  - IPC with address **106** in DIF **Layer11** is reachable via DIFs **Layer01**, **Layer02** and **Layer03** and IPCs with addresses **6**, **7** and **8**
  - IPC with address **107** in DIF **Layer11** is reachable via DIFs **Layer03**, **Layer04** and **Layer05** and IPCs with addresses **9**, **10** and **11**
  - IPC with address **108** in DIF **Layer11** is reachable via DIFs **Layer04**, **Layer06** and **Layer07** and IPCs with addresses **12**, **13** and **14**
- Neighbor table entries describing applications/IPC processes accesible via neighboring hosts
- All IPCP's resource allocators are setup with two available QoS-cubes

```
<?xml version="1.0"?>
<Configuration>
 <Host id="host1">
  <DA>
   <Directory>
```

```
    <APN apn="App1">
     <DIF difName="Layer11" ipcAddress="101" />
    </APN>
    <APN apn="App5">
     <DIF difName="Layer11" ipcAddress="105" />
    </APN>

    <APN apn="101_Layer11">
     <DIF difName="Layer01" ipcAddress="1" />
    </APN>
    <APN apn="105_Layer11">
     <DIF difName="Layer07" ipcAddress="5" />
    </APN>

    <APN apn="106_Layer11">
     <DIF difName="Layer01" ipcAddress="6" />
     <DIF difName="Layer02" ipcAddress="7" />
     <DIF difName="Layer03" ipcAddress="8" />
    </APN>
    <APN apn="107_Layer11">
     <DIF difName="Layer03" ipcAddress="9" />
     <DIF difName="Layer04" ipcAddress="10" />
     <DIF difName="Layer05" ipcAddress="11" />
    </APN>
    <APN apn="108_Layer11">
     <DIF difName="Layer04" ipcAddress="12" />
     <DIF difName="Layer06" ipcAddress="13" />
     <DIF difName="Layer07" ipcAddress="14" />
    </APN>
   </Directory>
   <NeighborTable>
    <APN apn="105_Layer11">
     <Neighbor apn="106_Layer11" />
    </APN>
   </NeighborTable>
  </DA>
</Host>

<Host id="host2">
 <DA>

 </DA>
</Host>

<Host id="host3">
 <DA>
```

```
 </DA>
</Host>

<Host id="host4">
 <DA>

 </DA>
</Host>

<Host id="host5">
 <DA>
  <NeighborTable>
   <APN apn="101_Layer11">
    <Neighbor apn="108_Layer11" />
   </APN>
  </NeighborTable>
 </DA>
</Host>

<Router id="router1">
 <DA>
  <Directory>
   <APN apn="App1">
    <DIF difName="Layer11" ipcAddress="101" />
   </APN>
   <APN apn="App5">
    <DIF difName="Layer11" ipcAddress="105" />
   </APN>

   <APN apn="101_Layer11">
    <DIF difName="Layer01" ipcAddress="1" />
   </APN>
   <APN apn="105_Layer11">
    <DIF difName="Layer07" ipcAddress="5" />
   </APN>

   <APN apn="106_Layer11">
    <DIF difName="Layer01" ipcAddress="6" />
    <DIF difName="Layer02" ipcAddress="7" />
    <DIF difName="Layer03" ipcAddress="8" />
   </APN>
   <APN apn="107_Layer11">
    <DIF difName="Layer03" ipcAddress="9" />
    <DIF difName="Layer04" ipcAddress="10" />
    <DIF difName="Layer05" ipcAddress="11" />
```

```
    </APN>
    <APN apn="108_Layer11">
     <DIF difName="Layer04" ipcAddress="12" />
     <DIF difName="Layer06" ipcAddress="13" />
     <DIF difName="Layer07" ipcAddress="14" />
    </APN>

   </Directory>
   <NeighborTable>
    <APN apn="105_Layer11">
     <Neighbor apn="107_Layer11" />
    </APN>
   </NeighborTable>
  </DA>
 </Router>
 <Router id="router2">
  <DA>
   <NeighborTable>
    <APN apn="105_Layer11">
     <Neighbor apn="108_Layer11" />
    </APN>
    <APN apn="101_Layer11">
     <Neighbor apn="106_Layer11" />
    </APN>
   </NeighborTable>
  </DA>
 </Router>
 <Router id="router3">
  <DA>
   <NeighborTable>
    <APN apn="101_Layer11">
     <Neighbor apn="107_Layer11" />
    </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>0</ForceOrder>
    <MaxAllowableGap>10</MaxAllowableGap>
    <Delay>1000000</Delay>
    <Jitter>500000</Jitter>
    <CostTime>0</CostTime>
    <CostBits>0</CostBits>
    <ATime>0</ATime>
   </QoSCube>
   <QoSCube id="2">
    <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>10</MaxAllowableGap>
    <Delay>1000000</Delay>
    <Jitter>500000</Jitter>
    <CostTime>0</CostTime>
    <CostBits>0</CostBits>
    <ATime>0</ATime>
   </QoSCube>
  </QoSCubesSet>
</Configuration>
```

5.4. All Nodes Example

5.4.1. Scenario

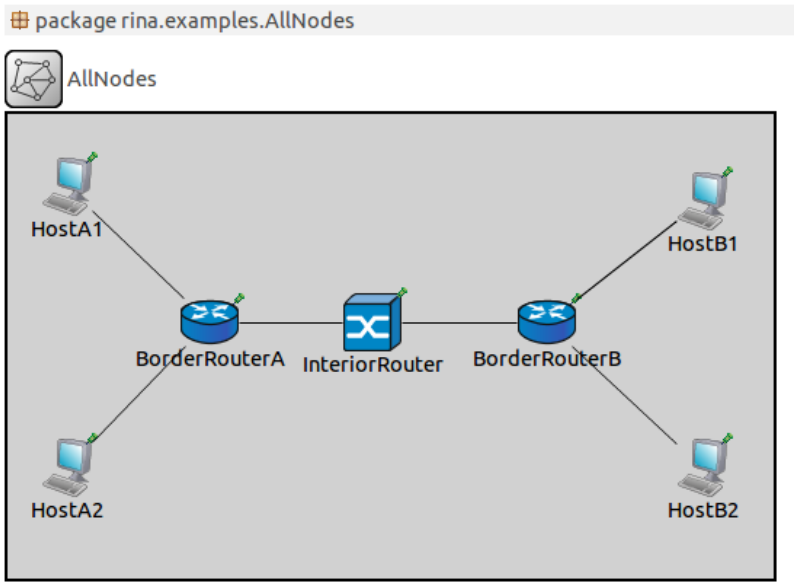

Figure 45. All Nodes Scenario

5.4.2. High-level components

4× `Host1AP.ned`, 2× `BorderRouter.ned`, 1× `InteriorRouter2Int.ned`

5.4.3. Simulation settings in omnetpp.ini

Settings contain the following setup of parameters:

- Used `AE` type is `AEPing`
- APs have assigned APN
- IPPCs are assigned an address and DIF name thus creating unique IPC APN
- DIF allocators are bound with static configuration of mappings
- Two ping scenarios exist. In each one, AP with `AEPing` on HostA1 is communicating with HostB1's `AEPing` AP

```
[General]
network = AllNodes

#Host AP config
**.HostA1.applicationProcess1.apName = "AppA1"
**.HostA2.applicationProcess1.apName = "AppA2"
```

```
**.HostB1.applicationProcess1.apName = "AppB1"
**.HostB2.applicationProcess1.apName = "AppB2"
**.applicationEntity.aeType = "AEPing"
**.iae.aeName = "Ping"

#Static DIF naming
**.Host*.ipcProcess1.difName   = "LayerX"
**.BorderRouter*.relayIpc.difName  = "LayerX"

**.HostA1.ipcProcess0.difName    = "LayerA1"
**.BorderRouterA.ipcProcess1.difName  = "LayerA1"
**.HostA2.ipcProcess0.difName    = "LayerA2"
**.BorderRouterA.ipcProcess2.difName  = "LayerA2"

**.HostB1.ipcProcess0.difName    = "LayerB1"
**.BorderRouterB.ipcProcess1.difName  = "LayerB1"
**.HostB2.ipcProcess0.difName    = "LayerB2"
**.BorderRouterB.ipcProcess2.difName  = "LayerB2"

**.BorderRouterA.ipcProcess3.difName  = "LayerAB"
**.InteriorRouter.relayIpc.difName   = "LayerAB"
**.BorderRouterB.ipcProcess3.difName  = "LayerAB"

**.BorderRouterA.bottomIpc.difName   = "LayerYA"
**.InteriorRouter.ipcProcess0.difName  = "LayerYA"
**.BorderRouterB.bottomIpc.difName   = "LayerYB"
**.InteriorRouter.ipcProcess1.difName  = "LayerYB"

#Static IPC Addressing
**.HostA1.ipcProcess1.ipcAddress   = "A1"
**.HostA2.ipcProcess1.ipcAddress   = "A2"
**.HostB1.ipcProcess1.ipcAddress   = "B1"
**.HostB2.ipcProcess1.ipcAddress   = "B2"
**.BorderRouterA.relayIpc.ipcAddress  = "BRA"
**.BorderRouterB.relayIpc.ipcAddress  = "BRB"

**.HostA1.ipcProcess0.ipcAddress   = "a1"
**.BorderRouterA.ipcProcess1.ipcAddress = "bra1"
**.HostA2.ipcProcess0.ipcAddress   = "a2"
**.BorderRouterA.ipcProcess2.ipcAddress = "bra2"

**.HostB1.ipcProcess0.ipcAddress   = "b1"
**.BorderRouterB.ipcProcess1.ipcAddress = "brb1"
**.HostB2.ipcProcess0.ipcAddress   = "b2"
**.BorderRouterB.ipcProcess2.ipcAddress = "brb2"
```

```
**.BorderRouterA.ipcProcess3.ipcAddress = "A"
**.InteriorRouter.relayIpc.ipcAddress = "Z"
**.BorderRouterB.ipcProcess3.ipcAddress = "B"

**.BorderRouterA.bottomIpc.ipcAddress   = "ya"
**.InteriorRouter.ipcProcess0.ipcAddress  = "yza"
**.BorderRouterB.bottomIpc.ipcAddress   = "yb"
**.InteriorRouter.ipcProcess1.ipcAddress  = "yzb"

#DIF Allocator settings
**.HostA1.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='HostA12']/DA")
**.HostA2.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='HostA12']/DA")
**.HostB1.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='HostB12']/DA")
**.HostB2.difAllocator.configData = xmldoc("config.xml", "Configuration/
Host[@id='HostB12']/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")

#Directory settings
**.HostA1.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='HostA12']/DA")
**.HostA2.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='HostA12']/DA")
**.HostB1.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='HostA12']/DA")
**.HostB2.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='HostA12']/DA")

**.BorderRouterA.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='HostA12']/DA")
**.BorderRouterB.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='HostA12']/DA")
**.InteriorRouter.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Host[@id='HostA12']/DA")

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

```
[Config Ping]
fingerprint = "478d-3ee3"

#PingApp setup
**.HostA1.applicationProcess1.applicationEntity.iae.dstApName = "AppB1"
**.HostA1.applicationProcess1.applicationEntity.iae.dstAeName = "Ping"
**.HostA1.applicationProcess1.applicationEntity.iae.startAt = 10s
**.HostA1.applicationProcess1.applicationEntity.iae.pingAt =  15s
**.HostA1.applicationProcess1.applicationEntity.iae.rate = 5
**.HostA1.applicationProcess1.applicationEntity.iae.stopAt = 30s
**.BorderRouterA.bottomIpc.efcp.efcp.pduDroppingEnabled = false
**.HostA1.applicationProcess1.applicationEntity.iae.forceOrder = true

[Config PingWithDrop]
fingerprint = "90f6-7ce6"

#PingApp setup
**.HostA1.applicationProcess1.applicationEntity.iae.dstApName = "AppB1"
**.HostA1.applicationProcess1.applicationEntity.iae.dstAeName = "Ping"
**.HostA1.applicationProcess1.applicationEntity.iae.startAt = 10s
**.HostA1.applicationProcess1.applicationEntity.iae.pingAt =  15s
**.HostA1.applicationProcess1.applicationEntity.iae.rate = 10
**.HostA1.applicationProcess1.applicationEntity.iae.stopAt = 50s
**.HostA1.ipcProcess1.efcp.efcp.pduDroppingEnabled = true
#**.BorderRouterA.bottomIpc.efcp.efcp.pduDroppingEnabled = true
**.HostA1.applicationProcess1.applicationEntity.iae.forceOrder = true
```

5.4.4. Static configuration in config.xml

The following configuration introduces:

- DIF Allocator Directory mappings for APs and IPCs
  - AP **App1** is reachable via DIF with name **Layer1** and IPC with address **11**
  - AP **App2** is reachable via DIF with name **Layer1** and IPC with address **22**
  - IPC with address **11** in DIF **Layer1** is reachable via DIF with name **Layer0** and IPC with address **1**
  - IPC with address **22** in DIF **Layer1** is reachable via DIF with name **Layer0** and IPC with address **2**
  - Synonym for **App2** is **AppErr**

- All IPC's resource allocators are setup with two available QoS-cubes

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

 <Host id="HostA12">
  <DA>
   <Directory>
    <APN apn="AppA1">
     <DIF difName="LayerX" ipcAddress="A1" />
    </APN>
    <APN apn="AppA2">
     <DIF difName="LayerX" ipcAddress="A2" />
    </APN>
    <APN apn="AppB1">
     <DIF difName="LayerX" ipcAddress="B1" />
    </APN>
    <APN apn="AppB2">
     <DIF difName="LayerX" ipcAddress="B2" />
    </APN>

    <APN apn="A1_LayerX">
     <DIF difName="LayerA1" ipcAddress="a1" />
    </APN>
    <APN apn="A2_LayerX">
     <DIF difName="LayerA2" ipcAddress="a2" />
    </APN>
    <APN apn="B1_LayerX">
     <DIF difName="LayerB1" ipcAddress="b1" />
    </APN>
    <APN apn="B2_LayerX">
     <DIF difName="LayerB2" ipcAddress="b2" />
    </APN>

    <APN apn="BRA_LayerX">
     <DIF difName="LayerA1" ipcAddress="bra1" />
     <DIF difName="LayerA2" ipcAddress="bra2" />
     <DIF difName="LayerAB" ipcAddress="A" />
    </APN>
    <APN apn="BRB_LayerX">
     <DIF difName="LayerB1" ipcAddress="brb1" />
     <DIF difName="LayerB2" ipcAddress="brb2" />
     <DIF difName="LayerAB" ipcAddress="B" />
    </APN>
```

```
    <APN apn="A_LayerAB">
     <DIF difName="LayerYA" ipcAddress="ya" />
    </APN>
    <APN apn="B_LayerAB">
     <DIF difName="LayerYB" ipcAddress="yb" />
    </APN>
    <APN apn="Z_LayerAB">
     <DIF difName="LayerYA" ipcAddress="yza" />
     <DIF difName="LayerYB" ipcAddress="yzb" />
    </APN>
   </Directory>
   <NeighborTable>
    <APN apn="A1_LayerX">
     <Neighbor apn="BRA_LayerX" />
    </APN>
    <APN apn="A2_LayerX">
     <Neighbor apn="BRA_LayerX" />
    </APN>
    <APN apn="B1_LayerX">
     <Neighbor apn="BRA_LayerX" />
    </APN>
    <APN apn="B2_LayerX">
     <Neighbor apn="BRA_LayerX" />
    </APN>
   </NeighborTable>
  </DA>
 </Host>

 <Host id="HostB12">
  <DA>
   <NeighborTable>
    <APN apn="A1_LayerX">
     <Neighbor apn="BRB_LayerX" />
    </APN>
    <APN apn="A2_LayerX">
     <Neighbor apn="BRB_LayerX" />
    </APN>
    <APN apn="B1_LayerX">
     <Neighbor apn="BRB_LayerX" />
    </APN>
    <APN apn="B2_LayerX">
     <Neighbor apn="BRB_LayerX" />
    </APN>
   </NeighborTable>
  </DA>
 </Host>
```

```
<Router id="BorderRouterA">
 <DA>
  <NeighborTable>
   <APN apn="B1_LayerX">
    <Neighbor apn="BRB_LayerX" />
   </APN>
   <APN apn="B_LayerAB">
    <Neighbor apn="Z_LayerAB" />
   </APN>
  </NeighborTable>
 </DA>
</Router>

<Router id="BorderRouterB">
 <DA>
  <NeighborTable>
   <APN apn="A1_LayerX">
    <Neighbor apn="BRA_LayerX" />
   </APN>
   <APN apn="A_LayerAB">
    <Neighbor apn="Z_LayerAB" />
   </APN>
  </NeighborTable>
 </DA>
</Router>

<Router id="InteriorRouter">
 <DA>
  <NeighborTable>
   <APN apn="A1_LayerX">
    <Neighbor apn="BRB_LayerX" />
   </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>0</ForceOrder>
    <MaxAllowableGap>10</MaxAllowableGap>
    <Delay>1000000</Delay>
    <Jitter>500000</Jitter>
    <CostTime>0</CostTime>
    <CostBits>0</CostBits>
    <ATime>0</ATime>
   </QoSCube>
   <QoSCube id="2">
    <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>10</MaxAllowableGap>
    <Delay>1000000</Delay>
    <Jitter>500000</Jitter>
    <CostTime>0</CostTime>
    <CostBits>0</CostBits>
    <ATime>0</ATime>
   </QoSCube>
  </QoSCubesSet>
 </Configuration>
```

5.5. Fat Tree Example

5.5.1. Motivation

This example introduce the DC Fat Tree topology with the application of dynamic routing. During the execution of this scenario you will be able to see the messages exchanged between IPCPs required to fill the routing table.

5.5.2. Scenario

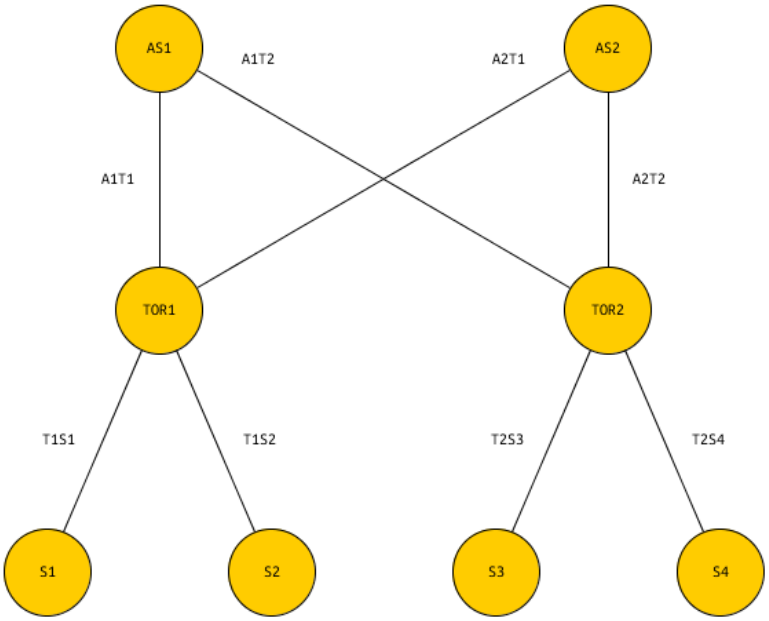

Figure 46. Fat Tree Scenario

5.5.3. High-level components

2× `Host1AP.ned` 4x `InteriorRouter4Int.ned` 2x `InteriorRouter2Int.ned`

5.5.4. Simulation settings in omnetpp.ini

- Used `AE` type is `AEPing`
- APs have assigned APN.
- IPCPs are assigned an address and DIF name thus creating unique IPC APN.
- DIF allocators are not bound with static configuration of mappings. The routes will be computed at runtime.* *
  - A smart summary of the information present in each node is showed during the simulation.
- One scenario exists:
  - FatTreeTopology scenario will start at second 130 a communication between AE1(on Server1) and AE3(on Server3).

```
[General]
```

```
network = FatTreeTopology
sim-time-limit = 5min
seed-set = ${runnumber}
sim-time-limit = 5min
seed-set = ${runnumber}
debug-on-errors = true

#
# Appliction entities naming:
#
**.Server1.applicationProcess1.apName = "App1"
**.Server2.applicationProcess1.apName = "App2"
**.Server3.applicationProcess1.apName = "App3"
**.Server4.applicationProcess1.apName = "App4"
**.applicationEntity.aeType = "AEPing"
**.iae.aeName = "Ping"

#
# Server instances addressing:
#
# Shims:
**.Server1.ipcProcess0.difName = "T1S1"
**.Server1.ipcProcess0.ipcAddress = "2"
# DataCenter wide DIF.
**.Server1.ipcProcess1.difName = "DC"
**.Server1.ipcProcess1.ipcAddress = "S1"

# Shims:
**.Server2.ipcProcess0.difName = "T1S2"
**.Server2.ipcProcess0.ipcAddress = "2"
# DataCenter wide DIF.
**.Server2.ipcProcess1.difName = "DC"
**.Server2.ipcProcess1.ipcAddress = "S2"

# Shims:
**.Server3.ipcProcess0.difName = "T2S3"
**.Server3.ipcProcess0.ipcAddress = "2"
# DataCenter wide DIF.
**.Server3.ipcProcess1.difName = "DC"
**.Server3.ipcProcess1.ipcAddress = "S3"

# Shims:
**.Server4.ipcProcess0.difName = "T2S4"
**.Server4.ipcProcess0.ipcAddress = "2"
# DataCenter wide DIF.
**.Server4.ipcProcess1.difName = "DC"
```

```
**.Server4.ipcProcess1.ipcAddress = "S4"

#
# TOR instances addressing:
#
# Shims to aggregators:
**.TOR1.ipcProcess0.difName = "A1T1"
**.TOR1.ipcProcess0.ipcAddress = "2"
**.TOR1.ipcProcess1.difName = "A2T1"
**.TOR1.ipcProcess1.ipcAddress = "2"
# Shims to servers:
**.TOR1.ipcProcess2.difName = "T1S1"
**.TOR1.ipcProcess2.ipcAddress = "1"
**.TOR1.ipcProcess3.difName = "T1S2"
**.TOR1.ipcProcess3.ipcAddress = "1"
# DataCenter wide DIF.
**.TOR1.relayIpc.difName = "DC"
**.TOR1.relayIpc.ipcAddress = "TOR1"

# Shims to aggregators:
**.TOR2.ipcProcess0.difName = "A1T2"
**.TOR2.ipcProcess0.ipcAddress = "2"
**.TOR2.ipcProcess1.difName = "A2T2"
**.TOR2.ipcProcess1.ipcAddress = "2"
# Shims to servers:
**.TOR2.ipcProcess2.difName = "T2S3"
**.TOR2.ipcProcess2.ipcAddress = "1"
**.TOR2.ipcProcess3.difName = "T2S4"
**.TOR2.ipcProcess3.ipcAddress = "1"
# DataCenter wide DIF.
**.TOR2.relayIpc.difName = "DC"
**.TOR2.relayIpc.ipcAddress = "TOR2"

#
# Aggregators instances addressing:
#
# Shims:
**.AS1.ipcProcess0.difName = "A1T1"
**.AS1.ipcProcess0.ipcAddress = "1"
**.AS1.ipcProcess1.difName = "A1T2"
**.AS1.ipcProcess1.ipcAddress = "1"
# DataCenter wide DIF.
**.AS1.relayIpc.difName = "DC"
**.AS1.relayIpc.ipcAddress = "AS1"

# Shims:
```

```
**.AS2.ipcProcess0.difName = "A2T1"
**.AS2.ipcProcess0.ipcAddress = "1"
**.AS2.ipcProcess1.difName = "A2T2"
**.AS2.ipcProcess1.ipcAddress = "1"
# DataCenter wide DIF.
**.AS2.relayIpc.difName = "DC"
**.AS2.relayIpc.ipcAddress = "AS2"

#
# Policy selection for DC Dif.
#
**.Server*.ipcProcess1.resourceAllocator.pduftgPolicyName =
 "DistanceVectorPolicy"
**.Server*.ipcProcess1.resourceAllocator.pduFwdTabGenerator.netStateVisible
 = true
**.Server*.ipcProcess1.resourceAllocator.pduFwdTabGenerator.netStateMod =
 "^.^.^"

**.TOR*.relayIpc.resourceAllocator.pduftgPolicyName =
 "DistanceVectorPolicy"
**.TOR*.relayIpc.resourceAllocator.pduFwdTabGenerator.netStateVisible =
 true
**.TOR*.relayIpc.resourceAllocator.pduFwdTabGenerator.netStateMod =
 "^.^.^"

**.AS*.relayIpc.resourceAllocator.pduftgPolicyName =
 "DistanceVectorPolicy"
**.AS*.relayIpc.resourceAllocator.pduFwdTabGenerator.netStateVisible =
 true
**.AS*.relayIpc.resourceAllocator.pduFwdTabGenerator.netStateMod = "^.^.^"

#
# DIF Allocator settings
#
**.Server1.difAllocator.configData = xmldoc("config.xml", "Configuration/
Switch[@id='AS1']/DA")
**.Server2.difAllocator.configData = xmldoc("config.xml", "Configuration/
Switch[@id='AS1']/DA")
**.Server3.difAllocator.configData = xmldoc("config.xml", "Configuration/
Switch[@id='AS1']/DA")
**.Server4.difAllocator.configData = xmldoc("config.xml", "Configuration/
Switch[@id='AS1']/DA")

**.TOR1.difAllocator.configData = xmldoc("config.xml", "Configuration/
Switch[@id='AS1']/DA")
```

```
**.TOR2.difAllocator.configData = xmldoc("config.xml", "Configuration/
Switch[@id='AS1']/DA")

**.AS1.difAllocator.configData = xmldoc("config.xml", "Configuration/
Switch[@id='AS1']/DA")
**.AS2.difAllocator.configData = xmldoc("config.xml", "Configuration/
Switch[@id='AS1']/DA")

#
# Directory settings
#
**.Server1.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Switch[@id='AS1']/DA")
**.Server2.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Switch[@id='AS1']/DA")
**.Server3.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Switch[@id='AS1']/DA")
**.Server4.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Switch[@id='AS1']/DA")

**.TOR1.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Switch[@id='AS1']/DA")
**.TOR2.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Switch[@id='AS1']/DA")

**.AS1.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Switch[@id='AS1']/DA")
**.AS2.difAllocator.directory.configData = xmldoc("config.xml",
 "Configuration/Switch[@id='AS1']/DA")

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

#
# Preallocated flow on hosts:
#
**.Server1.ipcProcess1.resourceAllocator.ra.flows = xmldoc("config.xml",
 "Configuration/Server[@id='Server1']/IPC[@id='ipcProcess1']/RA/Flows")
**.Server2.ipcProcess1.resourceAllocator.ra.flows = xmldoc("config.xml",
 "Configuration/Server[@id='Server2']/IPC[@id='ipcProcess1']/RA/Flows")
**.Server3.ipcProcess1.resourceAllocator.ra.flows = xmldoc("config.xml",
 "Configuration/Server[@id='Server3']/IPC[@id='ipcProcess1']/RA/Flows")
**.Server4.ipcProcess1.resourceAllocator.ra.flows = xmldoc("config.xml",
 "Configuration/Server[@id='Server4']/IPC[@id='ipcProcess1']/RA/Flows")
```

```
**.TOR1.relayIpc.resourceAllocator.ra.flows = xmldoc("config.xml",
 "Configuration/Switch[@id='TOR1']/IPC[@id='relayIpc']/RA/Flows")
**.TOR2.relayIpc.resourceAllocator.ra.flows = xmldoc("config.xml",
 "Configuration/Switch[@id='TOR2']/IPC[@id='relayIpc']/RA/Flows")

[Config FatTreeTopology]
fingerprint = "9be6-59a1"
#
# AEs don't do anything. We're only evaluating the routing table now.
#

**.Server1.applicationProcess1.applicationEntity.iae.dstApName = "App3"
**.Server1.applicationProcess1.applicationEntity.iae.dstAeName = "Ping"
**.Server1.applicationProcess1.applicationEntity.iae.startAt = 130s
**.Server1.applicationProcess1.applicationEntity.iae.pingAt = 140s
**.Server1.applicationProcess1.applicationEntity.iae.rate = 5
**.Server1.applicationProcess1.applicationEntity.iae.stopAt = 200s
```

5.5.5. Static configuration in config.xml

The following configuration introduces:

- Each Node will establish a flow, during the first step of the simulation, with the neighbors.
- The DIF Allocator of every node has the following information:
  - Aggregator Switches (AS) are connected with Top of Racks (TORS) through their own Shim DIFs(AnTm, where 'n' and 'm' are the AS and TOR number).
  - Top of Racks(TORs) have Shims to the connected server(TmSo, where 'm' and 'o' are the TOR and Server number).
  - Application Entities are connected with the Data Center DIF inside each Server node.
- The two standard QoS cube are available.

```
<?xml version="1.0"?>
<Configuration>
  <Server id="Server1">
    <IPC id="ipcProcess1">
      <RA>
        <Flows>
```

```
        <Flow apn="TOR1_DC" qosCube="1"/>
      </Flows>
    </RA>
  </IPC>
</Server>
<Server id="Server2">
  <IPC id="ipcProcess1">
    <RA>
      <Flows>
        <Flow apn="TOR1_DC" qosCube="1"/>
      </Flows>
    </RA>
  </IPC>
</Server>
<Server id="Server3">
  <IPC id="ipcProcess1">
    <RA>
      <Flows>
        <Flow apn="TOR2_DC" qosCube="1"/>
      </Flows>
    </RA>
  </IPC>
</Server>
<Server id="Server4">
  <IPC id="ipcProcess1">
    <RA>
      <Flows>
        <Flow apn="TOR2_DC" qosCube="1"/>
      </Flows>
    </RA>
  </IPC>
</Server>
<Switch id="TOR1">
  <IPC id="relayIpc">
    <RA>
      <Flows>
        <Flow apn="AS1_DC" qosCube="1"/>
        <Flow apn="AS2_DC" qosCube="1"/>
      </Flows>
    </RA>
  </IPC>
</Switch>
<Switch id="TOR2">
  <IPC id="relayIpc">
    <RA>
      <Flows>
```

```
          <Flow apn="AS1_DC" qosCube="1"/>
          <Flow apn="AS2_DC" qosCube="1"/>
        </Flows>
      </RA>
    </IPC>
  </Switch>
  <Switch id="AS1">
    <!--
      This contains the whole mapping of the network.
      It can be used to the Dif Allocator of every element.
    -->
    <DA>
      <Directory>
        <!--
          How the DIF name are formatted?
          They contain, for reading purposes, the initial letter of the
"upper"
          element in the simulation and the initial letter of the "bottom"
element.
          Example: A1T1 means Aggregator1 to Tor1.
        -->

        <!-- Aggregator side naming of the Shims -->
        <APN apn="AS1_DC">
          <DIF difName="A1T1" ipcAddress="1"/>
          <DIF difName="A1T2" ipcAddress="1"/>
        </APN>
        <APN apn="AS2_DC">
          <DIF difName="A2T1" ipcAddress="1"/>
          <DIF difName="A2T2" ipcAddress="1"/>
        </APN>

        <!-- TOR side naming of the Shims -->
        <APN apn="TOR1_DC">
          <DIF difName="A1T1" ipcAddress="2"/>
          <DIF difName="A2T1" ipcAddress="2"/>

          <DIF difName="T1S1" ipcAddress="1"/>
          <DIF difName="T1S2" ipcAddress="1"/>
        </APN>
        <APN apn="TOR2_DC">
          <DIF difName="A1T2" ipcAddress="2"/>
          <DIF difName="A2T2" ipcAddress="2"/>

          <DIF difName="T2S3" ipcAddress="1"/>
          <DIF difName="T2S4" ipcAddress="1"/>
```

```
      </APN>

      <!-- Server side naming of the Shims -->
      <APN apn="S1_DC">
        <DIF difName="T1S1" ipcAddress="2"/>
      </APN>
      <APN apn="S2_DC">
        <DIF difName="T1S2" ipcAddress="2"/>
      </APN>
      <APN apn="S3_DC">
        <DIF difName="T2S3" ipcAddress="2"/>
      </APN>
      <APN apn="S4_DC">
        <DIF difName="T2S4" ipcAddress="2"/>
      </APN>

      <!-- AE side naming of the DC IPCs -->
      <APN apn="App1">
        <DIF difName="DC" ipcAddress="S1"/>
      </APN>
      <APN apn="App2">
        <DIF difName="DC" ipcAddress="S2"/>
      </APN>
      <APN apn="App3">
        <DIF difName="DC" ipcAddress="S3"/>
      </APN>
      <APN apn="App4">
        <DIF difName="DC" ipcAddress="S4"/>
      </APN>
    </Directory>
  </DA>
</Switch>
<Switch id="AS2">
</Switch>
<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>0</ForceOrder>
      <MaxAllowableGap>10</MaxAllowableGap>
      <Delay>1000000</Delay>
      <Jitter>500000</Jitter>
      <CostTime>0</CostTime>
      <CostBits>0</CostBits>
      <ATime>0</ATime>
    </QosCube>
    <QosCube id="2">
      <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>10</MaxAllowableGap>
      <Delay>1000000</Delay>
      <Jitter>500000</Jitter>
      <CostTime>0</CostTime>
      <CostBits>0</CostBits>
      <ATime>0</ATime>
    </QosCube>
  </QoSCubesSet>
</Configuration>
```

5.5.6. Scenario description

Scenario FatTreeTopology has the following phases:

- At t=0 the nodes pre-allocate flows with the neighbors.
- Every 30 seconds(default routing policy timeout) the nodes exchanges routing information between themselves, updating their touring table.
- At t=130 `AE1`, located in `Server1`, begins a ping communication with `AE3`, located in `Server3`.

6. Conclusions

This report described the RINASim status as of M13 of PRISTINE project. It contains information on how to obtain, install, configure, modify and run RINASim components within the OMNeT environment. The RINASim architecture and all major simulation blocks of RINASim are described to provide information to those who want to further extend RINASim with more features or implement their own policies that may be plugged in the architecture. The report also provided the description of demonstration examples on RINASim applications. Trying these examples, the user should be able to gain skills enabling her to create her own RINASim experiments. Though not complete the current RINASim version provides solid foundations for modeling and experimenting with different RINASim policies, e.g., policies for routing, security or data transfer. During upcoming months, we will extend RINASim in several directions: a) to overcome the current limitations of RINASim models that provide only basic mechanisms in many areas; b) to incorporate additional features based on requirements emerged from results of other WPs and c) to add new models and improve current models according to partners suggestions. We plan to deliver the next version of RINASim in M23 with a couple of previews enabling to evaluate the RINASim within the consortium during the development period.

References

- [omnetpp-dwnld] OpenSim Ltd., OMNeT++ Releases, available online[3]
- [ops-rinasimtickets] OpenSource Projects, RINASim Tickets, available online[4]
- [github-kvetak] GitHub, RINA Simulator repository, available online[5]
- [omnetpp-main] OpenSim Ltd., OMNeT++ Discrete Event Simulator, available online[6]
- [omnetpp-inet] OpenSim Ltd., INET Framework, available online[7]
- [omnetpp-ansa] OpenSim Ltd., ANSA Project, available online[8]
- [omnetpp-mixim] OpenSim Ltd., MIXIM Framework, available online[9]
- [omnetpp-oversim] OpenSim Ltd., Oversim Framework, available online[10]
- [omnetpp-veins] OpenSim Ltd., Veins Framework, available online[11]
- [omnetpp-castalia] OpenSim Ltd., Castalia Framework, available online[12]
- [omnetpp-manual] OpenSim Ltd., Manual, available online[13]
- [omnetpp-ide] OpenSim Ltd., IDE in Nutshell, available online[14]
- [omnetpp-eclipse] OpenSim Ltd., Eclipse, available online[15]

[3] http://www.omnetpp.org/omnetpp/category/30-omnet-releases
[4] https://opensourceprojects.eu/p/pristine/rinasimulator/tickets/
[5] https://github.com/kvetak/RINA
[6] http://www.omnetpp.org
[7] http://inet.omnetpp.org/
[8] http://nes.fit.vutbr.cz/ansa
[9] http://mixim.sourceforge.net/
[10] http://www.oversim.org/
[11] http://veins.car2x.org/
[12] http://castalia.research.nicta.com.au/index.php/en/
[13] http://www.omnetpp.org/doc/omnetpp/manual/usman.html
[14] http://www.omnetpp.org/pmwiki/index.php?n=Main.OmnetppInNutshell
[15] http://www.omnest.com/webdemo/ide/demo.html