Ref. Ares(2015)3645348 - 04/09/2015


# Deliverable-2.3

## Proof of concept of the Software Development Kit

Deliverable Editor: Francesco Salvestrini, Nextworks s.r.l. (NXW)


| | |
|---|---|
| Publication date: | 31-January-2015 |
| Deliverable Nature: | 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: | Software Development Kit, Policy sets, RINA Plugin Infrastructure, Programmability, Application Programming Interfaces, open IRATI RINA implementation |
| Synopsis: | This document describes the initial proof of concept of PRISTINE's Software Development Kit for the Open IRATI stack. |

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

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

### List of Contributors

Deliverable Editor: Francesco Salvestrini, Nextworks s.r.l. (NXW)
nxw: Vincenzo Maffione, Francesco Salvestrini
i2cat: Eduard Grasa
atos: Miguel Angel
bisdn: Marc Sune
tssg: Miguel Ponce de Leon, Micheal Crotty
iminds: Sander Vrijders
fit-but: Ondrej Rysavy
upc: Jordi Perello
uio: Michael Welzl

### Disclaimer

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

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

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

# Executive summary

This deliverable describes the initial version of the Software Development Kit (SDK) for the open source RINA implementation called IRATI. T2.3 has performed an initial review of the open IRATI software, defined a framework that allows to plug and unplug developer-defined policies through a set of Application Programming Interfaces (APIs), designed mechanisms and procedures that allow for the insertion and removal of such policies at run-time and finally implemented the APIs for a subset of the policies available in the DIF (the subset of APIs implemented has been chosen based on the research and experimentation needs of the WP3, WP4 and WP5 partners).

The PRISTINE SDK facilitates the programmability of the policies in a DIF, easing their development and reducing the implementation effort at minimum. Policy developers only need to implement the policies they are interested in - e.g. a new routing algorithm, a new authentication strategy, a different resource allocation policy - adapting them to the API provided by the SDK. Therefore, the PRISTINE SDK can be positioned as an evolution of the IRATI code base that incrementally introduces hooks where existing policies used to be hardwired in the code, introduces the missing software components embedded with the programmability-enabling mechanisms and enhances the whole code-base for the project's scopes. Before the SDK under development by PRISTINE, a developer needed to understand the IRATI codebase, modify the sources and recompile all the code in order to introduce new policies. After the SDK, developers only need to implement a number of APIs and package the implementation classes as a plugin which can be dynamically loaded into a running IRATI stack.

This deliverable illustrates the design decisions and high level architecture of the SDK for the project's first phase. In the next months, in terms of WP6 activities, the SDK will be integrated with the policies developed by the project's partners and further stabilised given the broader range of tests which will be handled within WP6. Feedbacks collected during the integration activities will be taken into account for the next iteration, which will also provide additional programmability extensions to the SDK delivered as part of D2.5.

# Table of Contents

# Glossary

## 1. 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.

## 2. 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.

**CACEP**
Common Application Connection Establishment Phase.

**CDAP**
Common Distributed Application Protocol.

**CLI**
Command Line Interface.

**CMIP**
Common Management Information Protocol.

**CRC**
Cyclic Redundancy Code.

**DAF**
Distributed Application Facility.

**DAP**
Distributed Application Process.

**DMS**
DIF Management System.

**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.

**IPC**
Inter Process Communication.

**IRM**
IPC Resource Manager.

**JSON**
Java Script Object Notation.

**JVM**
Java Virtual Machine.

**KRPI**
Kernel space RINA Plugins Infrastructure.

**LKM**
Loadable Kernel Module.

**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.

**OOD**
Object Oriented Development.

**OOP**
Object Oriented Programming.

**OS**
Operating System.

**PCI**
Protocol Control Information.

**PDU**
Protocol Data Unit.

**PM**
Protocol Machine.

**QoS**
Quality of Service.

**RA**
Resource Allocator.

**RAD**
Rapid Application Development.

**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

Pre-IRATI prototypes (e.g. Boston University's ProtoRINA [protorina]) principally focus on the validation of the RINA architecture, provide limited functionalities and adopt software designs that aim at facilitating experimentation. They are closed sources, completely implemented in user-space and make use of high-level programming languages such as Java. Therefore, they are also constrained in performances and implicitly inherit the limitations of the underlying Operating System (OS) - e.g. RINA can lay on a very limited set of shim DIFs, such as the shim DIF for TCP/IP.

The FP7-IRATI project [irati-home] addressed such concerns, designing and implementing from scratch a RINA software stack that has been released as open-source software [open-irati]. The IRATI stack (also known as the Open IRATI stack) provides core components of the RINA architecture for Linux based OS, it is implemented in C/C++ and lays its software architecture between user and kernel spaces. The FP7-IRATI project focused on the stack's software architecture in order to obtain an highly extensible solution. Since its first release [irati-d31], the architecture remained stable and the software has been updated with incremental additions that stack on top of each successive release, following a continuous development and integration approach.

Starting from the IRATI codebase, the PRISTINE SDK has been developed as a framework that allows for run-time policy plug-in and dynamic policy selection. While adding several new functionalities to the baseline IRATI stack, the SDK does not modify the IRATI High Level Architecture, which therefore remains stable with reference to [irati-d34].

PRISTINE's SDK has been released as open-source software, and is publicly available at the GitHub repository [open-irati-stack], the same public repository used for the FP7-IRATI software. The SDK codebase for the project's first phase is contained in a separate branch, pristine-1.1, whereas the master branch contains the last release of the IRATI software (described in [irati-d34]). Future SDK versions will be released by WP6 and WP2 as separate branches - e.g pristine-1.2, pristine-1.3 - allowing the project to evolve while keeping track of the status of the implementation in the different phases.

This document is structured as follows. Section 2 aims at providing an highlight of the IRATI stack's software architecture - a very brief summary of FP7 IRATI's deliverable [irati-d31], [irati-d32], [irati-d33] and [irati-d34] - which is the prerequisite for the comprehension of the following sections. Section 3 describes the requirements for

the policies of the first phase, accordingly to the PRISTINE Reference Framework [pristine-d22]. Section 4 introduces the methodologies that can be used to address the dynamic loading of code into the IRATI stack. Section 5 describes the high level architecture of the SDK, which stacks on top of the IRATI's software architecture. Section 6 provides details on interfacing the stack's code to high level languages (e.g. Java, Python), in order to allow interested partners to experiment with policies written in those languages. Section 7 reports the implementation status of the SDK, accordingly to the pristine-1.1 release. Finally, section 8 provides conclusions and future works.

# 2. The IRATI stack

The IRATI prototype provides an implementation of the RINA architecture - i.e. the IPC model - for a Linux-based Operating Systems, organized as a set of software packages that contain the different components used to implement RINA. The IPC Process functionalities have been partitioned between user and kernel spaces in order to enable the prototype to achieve and adequate level of performance and functionality. As an example, kernel software components - the shim IPC Processes - are necessary to directly access the network devices layer in order to allow RINA to be used on top of networking technologies such as Ethernet.

As illustrated in the Figure below, kernel components implement the data transfer and data transfer control parts of the IPC Process (Delimiting, the Error and Flow Control Protocol, the Relaying and Multiplexing Task and SDU Protection) and the shim IPC Processes. The layer management functions of the IPC Process and the local IPC Manager are implemented in user-space.

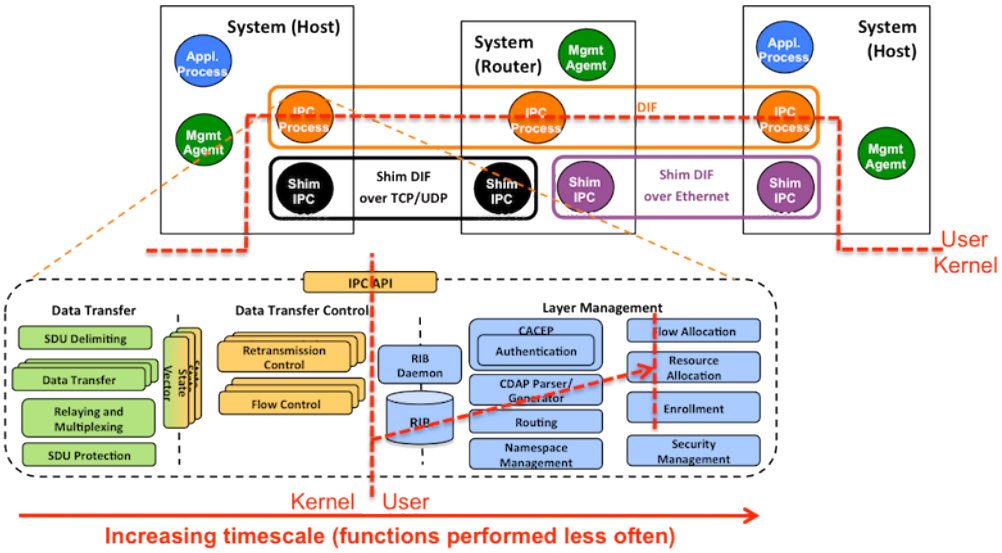


**Figure 1. Partitioning of functionalities between the user and kernel spaces**

User-space components provide the management layer related functionalities through a well defined set of user-space libraries and OS processes. The libraries wrap the kernel-space APIs (syscalls and Netlink messages) and provide additional functionalities such as:

- Allow applications to use RINA natively, enabling them to allocate and deallocate flows, read and write SDUs to these flows, and register/unregister to one or more DIFs.

- Assist the IPC Manager to perform the tasks related to IPC Process creation, deletion and configuration.
- Allow the IPC Process to configure the PDU forwarding table, create and delete EFCP instances, request the allocation of kernel resources to support a flow, etc.

The libraries - written in C/C - allow IRATI adopters to develop native RINA applications. In addition to that, language bindings to interpreted languages (i.e. Java) have been made available by wrapping the exported symbols with the target language native interface (i.e. JNI for Java). The IRATI OS daemons - the IPC Process and the IPC Manager - have been developed as C applications, ready to be used for testing and experimentation purposes.

The prototype provides a framework for configuration and building tasks, based on common and well estabilished tools. In particular, Linux [kconfig] and [kbuild] are used for kernel parts, while the autotools suite ([autoconf], [automake], [libtool]) is used for user-space parts. The prototype configuration framework automatically adapts to different OS/Linux based systems (e.g. Debian, Fedora, Ubuntu, Archlinux), and allows to build the stack with no user intervention.

Refer to [irati-ieee-network-magazine] for an introduction to the IRATI stack. Further details on the IRATI's stack high level architecture can be found in [irati-d21] and [irati-d23]. For details on the software design, refer to [irati-d31], [irati-d32], [irati-d33] and [irati-d34].

The following sections provide an overview of the IRATI software architecture.

## 2.1. User space architecture

The user-space components of the IRATI stack implement the layer management parts of the IPC Processes, as well as the DIF Allocator, IRM and Management Agent components of the RINA architecture reference model.

These components, depicted in the following figure, may be summarized as follows:

- Application process: An application that uses the RINA services to communicate with other applications, using the native RINA API.
- IPC Manager daemon: Local, system-wide manager of the RINA software. It is in charge of creating, configuring and destroying the other components of the RINA software - i.e. the IPC Processes. Interaction with the IPC Manager can be performed locally through a Command Line Interface (CLI) and configuration files, or remotely through a DIF Management System (DMS). The IPC Manager is also in charge

of brokering application registration and flow allocation requests/responses, by redirecting these operations to the most appropriate IPC Process Daemon. Finally, the IPC Manager also implements the DIF Allocator, which allows for the discovery of applications in DIFs the system is not a member of. There is a single instance of the IPC Manager daemon in each system.

- IPC Process daemon: Each IPC Process daemon implements the layer management components of an IPC Process (Flow Allocator, RIB Daemon, RIB, Resource Allocator, Enrollment Task, PDU Forwarding Table Generator). There is one instance of IPC Process daemon per each IPC Process in the system.

These high-level components rely on a common framework, implemented by the librina library. This library can be imagined as split in multiple libraries, providing all its functionalities as a whole. One of the possible partitioning may be the following:

- librina-application: Provides the APIs that allow an application to use RINA natively, enabling it to allocate and deallocate flows, read and write SDUs to that flows, and register/unregister to one or more DIFs. Under the hood it uses system calls to communicate with the kernel components or Netlink sockets to communicate with the user-space components. This library is used by regular applications, but also by the DIF Allocator, the IPC Processes Daemons and the IPC Manager (in its role of management agent, to communicate with the DMS process).
- librina-cdap: Provides the CDAP protocol functionalities.
- librina-common: Provides the common functionalities and definitions that must be shared among all other libraries and components.
- librina-utils: Provides the RINA related remaining functionalities not stated elsewhere in this list.

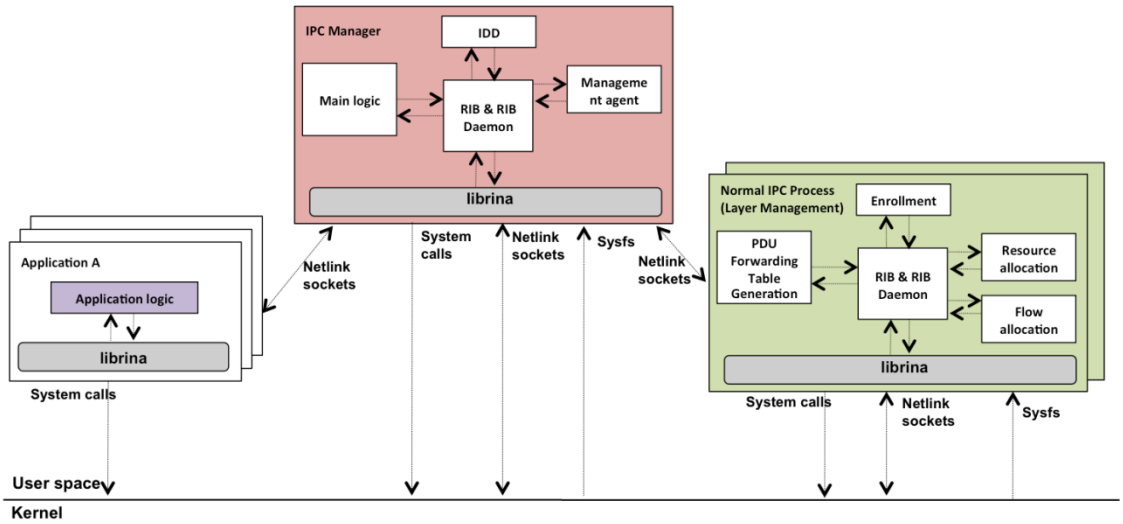


**Figure 2. The IRATI user-space high level architecture [irati-ieee-network-magazine]**

## 2.2. Kernel space architecture

The kernel-space parts of the IRATI's stack provide their functionalities through the EFCP, RMT, the PDU Forwarding table software components as well as different shim IPC Processes (e.g. the shim IPC Process over Ethernet for VLANs, the shim IPC Process over TCP/UDP, the shim IPC Process for Hypervisors).

These components are bound together by the KIPCM (the kernel IPC manager), the KFA (the Kernel Flow Allocator manager) and the RNL (the Netlink manager) layers that also implement the kernel/user interface.

The major software components in kernel-space are:

- Kernel IPC Manager (KIPCM): Manages the lifetime (creation, destruction, monitoring) of the other component instances in the kernel, as well as their configuration. It also provides coordination at the boundary among the different IPC Processes and applications. In the outgoing direction, it passes PDUs to an EFCP instance or to shim IPC Process. In the incoming direction, it passes PDUs to either an RMT instance or to an application process running in user space.
- Error and Flow Control Protocol (EFCP): Container of the different EFCP instances that implement the DTP and DTCP protocols for the connections supported by the IPC Processes. It passes PDUs to an RMT instance in the outgoing direction, or to the KIPCM in the incoming direction.
- Relaying and multiplexing task (RMT): Container of the different RMT instances in the system. Each RMT instance (one per IPC Process) multiplexes the PDUs generated by N EFCP connections to M underlying flows, and relies the PDUs coming from underlying flows to EFCP connections (local delivery) or to other underlying flows (forwarding). It passes PDUs to the KIPCM in the outgoing direction, and to EFCP in the incoming direction.
- SDU Protection: Container of the SDU Protection instances in the system. There can be a different SDU Protection instance associated to each N-1 port used by each IPC Process. SDU Protection policies include error detection and correction schemes, PDU lifetime enforcement mechanisms and encryption and compression policies.
- Shim IPC Process over Ethernet: Container of all the shim IPC Process over Ethernet instances in the system. This shim IPC Process is a "stripped-down" version of the IPC Process that wraps a 802.1q layer with the IPC Process interface. In the outgoing direction, this component passes PDUs to the relevant device driver. In the incoming direction, PDUs are extracted from received Ethernet frames passed to the KIPCM.

- Shim IPC Process over TCP/UDP: Container of all the shim IPC Process over TCP/UDP instances in the system. This shim IPC Process allows a normal DIF to be overlaid over an IP network using TCP and UDP as an underlying transport mechanism. It adapts the IP network to the shim DIF interface, mapping flow requests from upper layer IPC Processes to TCP or UDP sockets (depending on the requested Quality of Service). The shim IPC Process over TCP/UDP interacts with the sockets layer from within the Linux kernel.

The following figure depicts the kernel-space high-level architecture:

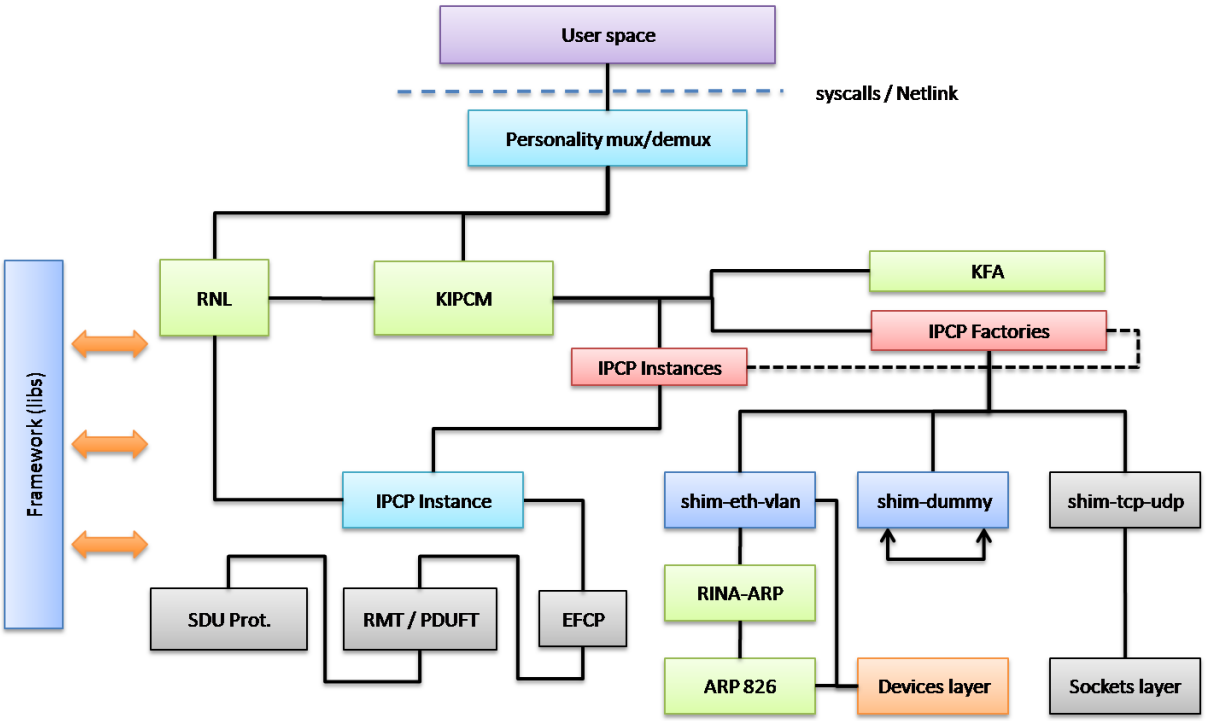


**Figure 3.**

# 3. Policy requirements

The following subsections provide information about the policies that are being targeted by the different PRISTINE technical WP during the project, grouped according to the IPC Process component they apply to. Some of the particular solutions for these policies will first be simulated and later implemented in the RINA prototype, as the research in the technical WP matures. The SDK must gradually provide support for developing different implementations of the following policies during the project lifetime. In fact, most of the policies described here are already supported by the current version of the SDK, as detailed in Section 7, "Implementation status".

## 3.1. Error and Flow Control Protocol (EFCP)

1. *Congestion Control* research area is interested in the following policies:
   a. The **set of flow control policies** that influence the rate at which EFCP can deliver PDUs to the RMT. The initial focus is on window-based flow control policies.
   b. **Transmission control policy**. The algorithm used to compute the new window size (or sending rate) based on the information on the ECN bits and other values recorded in the EFCP state vector.
   c. In order for the other EFCP policies to work properly, it is key to have a good **Round Trip Time (RTT) estimator policy**.

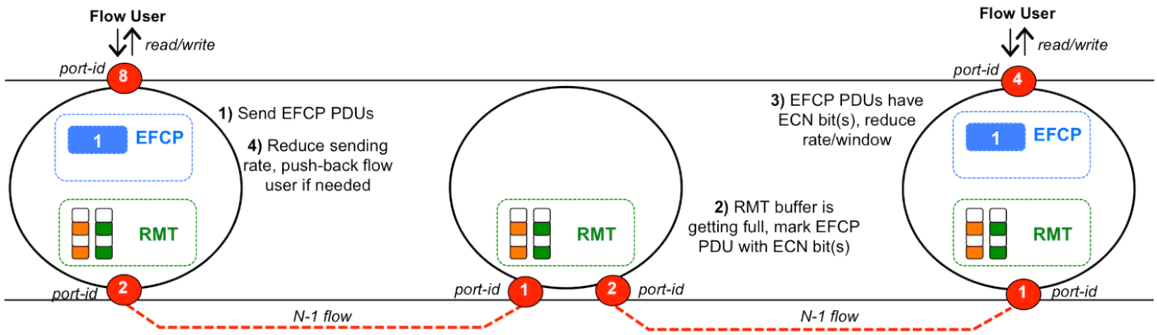


**Figure 4. Dealing with congestion in a DIF**

## 3.2. Relaying and Multiplexing Task (RMT)

1. *Congestion Control* research area is interested in the following policies:
   a. **RMT queue monitor policy**. To detect how the queues in the RMT are evolving and mark the PDUs with ECN-marks once the queue occupancy reaches a certain level.

2. *Resource Allocation* research area is interested in the following policies:

   a. **Input PDU classifier policy**. Takes incoming PDUs and classifies them among the different input queues. Assuming one input queue per QoS class, QoS class info would be extracted from each PDU's header and stored to the specific queue accordingly.

   b. **RMT queue monitor policy**.

      i. Input. Keeps track of the occupancy of the different input queues and the rate at which each QoS class is storing PDUs in the queues.

      ii. Output. Keeps track of the occupancy of the different output queues. Stores an array of queues, one for each Urgency level specified at the matrix of QoS classes. Each of these queues stores pointers to the actual output queues. The RMTSchedulingPolicy asks the RMTQMonitorPolicy the output queue from which the next PDU should be served. Upon that, the queue pointer at the most Urgent queue is returned. This allows decoupling Urgencies from the actual output queues, e.g., to perform QoS degradation of specific QoS classes efficiently.

   c. **Max queue policy**.

      i. Input. Keeps the occupancy of the different output queues. Stores an array of queues, one for each Urgency level specified at the matrix of QoS classes. Each of these queues stores pointers to the actual output queues. The RMTSchedulingPolicy asks the RMTQMonitorPolicy the output queue from which the next PDU should be served. Upon that, the queue pointer at the most Urgent queue is returned. This allows decoupling Urgencies from the actual output queues, e.g., to perform QoS degradation of specific QoS classes efficiently.

      ii. Output. Based on the measurements gathered by the RMTQMonitorPolicy, it can: 1) Drop the incoming PDU if the sum of the occupancies of all output queues to the N-1 output port has reached the cherish level for the QoS class the PDU belongs; 2) Drop the incoming PDU with certain probability if the total occupancy of all output queues is approaching (but still has not reached) the cherish level for that QoS class; 3) In a similar situation as in 2), degrade the QoS class of the next PDU to be served of that QoS class (this means that the next PDU of that class will be served with lower Urgency, i.e., a pointer to the queue assigned to that class put in a queue with lower Urgency in the RMTQMonitorPolicy).

   d. **RMT Scheduling policy**.

i. Input. Take PDUs from input queues and send them to the RMT (forwarding policy). Here, spacing between PDUs of a certain QoS class can be added, e.g., to avoid starvation of low priority output queues.

ii. Output. Requests to the RMTQMonitorPolicy the output queue from which the next PDU should be served, once the respective output port is available.

3. *Routing and Addressing* research area is interested in the following policies:
   a. **PDU Forwarding Policy**. Given the information in the header of the EFCP PDU and maybe some internal state of the policy (such a PDU Forwarding Table, for example), the PDU Forwarding policy returns one or more N-1 ports through which the EFCP PDU has to be forwarded. PDU Forwarding Policies based on a PDU Forwarding table computed by a routing policy (e.g. link-state or distance-vector) are envisioned, but also other types of policies that may be based on topological routing and may not require the computation of any PDU Forwarding table.
4. *Resource Allocation* research area is interested in the following policies:
   a. **PDU Forwarding Policy**. Design and implementation of a PDU Forwarding policy that exploits multiple N-1 flows available to reach the same destination (similar concept to ECMP, Equal Cost Multi-Path).

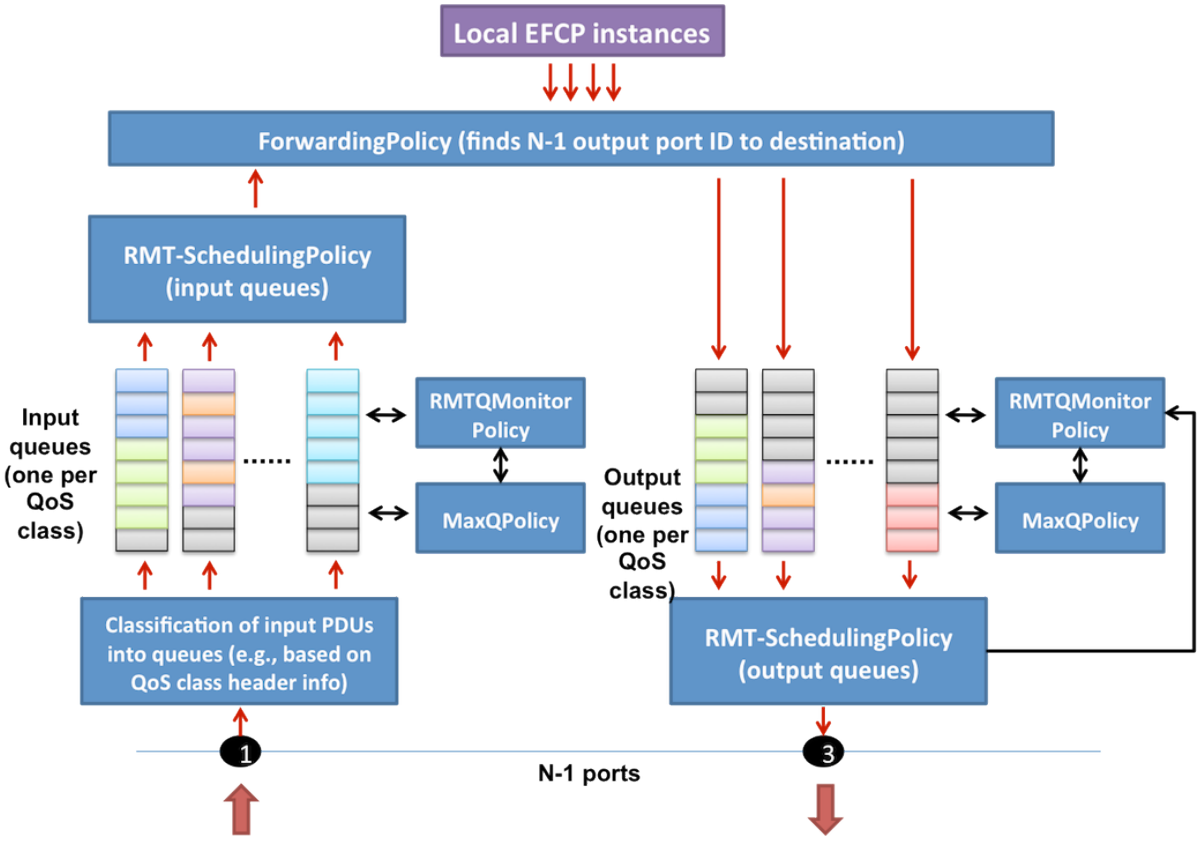


**Figure 5. Use of RMT policies to implement the cherish/urgency multiplexing approach**

## 3.3. Resource Allocator (RA)

1. *Resource Allocation* research area is interested in the following policies:
    a. **PDU forwarding table generator policy**. This policy takes the computation produced by routing as an input, as well as other information of the state of the IPC Process (such as the available N-1 ports, occupation of the RMT queues, etc.) and populates the PDU forwarding table. T3.2 is considering policies that leverage multiple routes with equal or different costs, spreading output PDUs belonging to different flows through multiple paths.

## 3.4. Routing

1. *Routing and Addressing* research area is interested in the following policies:
    a. **Routing policy**. Routing is all policy. Several approaches are possible and envisioned: link-state or distance-vector routing policies which generate a PDU Forwarding table; reactive routing, where routes are computed on demand; topological routing, where only special routes are computed (like short-cuts, routes to route around failures), but default routes are computed on the fly by the PDU Forwarding policy.
2. *Resource Allocation* research area is interested in the following policies:
    a. **Routing policy**. Multi-path routing policy that allows for the exploitation of multiple N-1 flows in order to reach the same destination.
3. iMinds, Nextworks, FIT-BUT, BISDN are interested in *Resiliency and High Availablility* research area. It is required the ability to customize the following policies:
    a. **Routing policy**. Per-hop link-state resilient routing policy, based on Loop-Free Alternates.

## 3.5. Flow Allocator (FA)

1. *Resiliency and High Availablility* research area is interested in the following policies:
    a. **Flow monitoring policy**.
        i. Flow Liveness Detection detects if a flow between IPC processes is alive or not by sending periodic messages. When FLD is present, the Flow Manager keeps two additional states for the flow - i.e. UP and DOWN. FLD maintains

a timer that is reset upon reception of such a periodic message. The flow is declared DOWN if the timer expires, otherwise it is declared UP.

ii. The Flow Loopback Request (FLR) should be executed only under error conditions. The procedure is activated by sending a CDAP M_START message to a neighbour IPC process containing the port-id to identify the flow to test. When an M_START_R is received back with a positive answer, all PDUs sent on that flow are looped back by the peering IPC process in order to asses the QoS level of the flow. After the monitoring traffic, an M_STOP CDAP message is sent to the neighbour IPC process. When it replies with a positive M_STOP_R, normal operations continue.

## 3.6. NameSpace Manager (NSM)

1. *Routing and Addressing* research area is interested in the following policies:

   a. **Address assignment policy**, **Address validation policy**. The address assignment and validation policies that will be considered for the first phase of the project are the following ones:

      i. *Centralized*. The NSM is structured in a centralized way, in which one or more IPC Process (or the Network Management System) maintain the full state of the address namespace within a DIF and manage address assignment in a centralized way.

      ii. *Hierarchical*. The DIF is structured in a hierarchy, breaking the DIF in subsets of IPC Processes which form clusters, clusters of clusters, etc. Each DIF subset is assigned a partition of the address space. One or more IPC Processes in each subset (for example, those IPC Processes belonging to edge routers) manage the address namespace for all the IPC Processes that belong to the subset.

## 3.7. SDU protection

1. *Authentication, Access control and Confidentiality* research area is interested in the following policies:

   a. **Encryption policy**. An encryption policy that extracts elements of the Transport Layer Security (TLS) record protocol is in scope for phase 1.

## 3.8. Security Manager

1. *Authentication, Access control and Confidentiality* research area is interested in the following policies:
    a. **Session negotiation policy**. Executed after authentication has successfully completed, this policy allows two peer IPC Processes to negotiate several items of the security context associated to the application connection (for example, the keys that need to be used for encryption or when to start encrypting messages).
    b. **New member access control policy**. Decides whether an authenticated IPC Process can join the DIF (based on capabilities).
    c. **RIB access control policy**. Decides whether an operation on an object in the RIB is accepted (based on capabilities).

## 3.9. CACEP

1. *Authentication, Access control and Confidentiality* research area is interested in the following policies:
    a. **Authentication policy**. Password-based authentication and certificate-based authentication, as defined in D4.1 are in scope for phase 1.

# 4. Plug-ins implementation methodologies

The RINA architecture defines a clear separation between mechanism and policy. Mechanism can be defined as the invariant and common part of any IPC dialogue within the RINA architecture, while policy is the configurable behavior that can be adapted to the particular scenario where IPC needs to take part.

The RINA architecture specification defines the different policy points where the IPC Process behaviour can be configured/adapted. A RINA implementation needs therefore to provide a way to "plug-in" certain pieces of code - the policies - on a per-IPCP basis.

The following sections describe various methodologies that could be used to implement the plug-in functionalities starting from the FP7-IRATI stack.

## 4.1. User-space plug-ins

User-space plug-ins can be implemented through the use of dynamic libraries and interpreted languages extensions, as explained in the following paragraphs.

### 4.1.1. Dynamic libraries

The shared objects technology is used, among other things, to provide the so-called “plug-in system”. A plug-in system allows the application to load (and link) compiled code at runtime, in order to provide additional features, not present in the original binary.

To implement plug-in systems, it is usually needed to call the dynamic linker at runtime, asking it to load the plug-in’s shared object. This object might just be a standard shared object or might require further details to be taken into consideration.

The call into the dynamic linker also varies for what concerns interface and implementation. Since most UNIX-like systems provide this interface through the dlopen() function, which is pretty much identical among them, lots of software rely on just this interface, and leave to [libtool] the task of building the plugins.

Software that needs to have a wider portability among different operating systems can use the wrapper library and interface called libltdl.

#### Creating libraries of C++ code

Creating libraries of C++ code is fairly straightforward process, because its object files differ from C ones in only three aspects:

- Name mangling: C++ libraries are only usable by the C++ compilers that created them. This decision was made by the designers of C++ in order to protect users from conflicting implementations of features such as constructors, exception handling, and RTTI.
- Dynamic initializers: on some systems, the C++ compiler must take special actions for the dynamic linker to run dynamic (i.e. run-time) initializers. This means that we should not call ld directly to link such libraries, and we should use the C++ compiler instead.
- Linking: C++ compilers will link some Standard C++ library in by default, but libtool does not know which are these libraries, so it cannot even run the inter-library dependence analyzer to check how to link it in. Therefore, running ld to link a C++ program or library is deemed to fail.

Because of these three issues, Libtool has been designed to always use the C++ compiler to compile and link C++ programs and libraries. In some instances the main() function of a program must also be compiled with the C++ compiler for static C++ objects to be properly initialized.

### 4.1.2. Interpreted language extensions

Almost all interpreted or JIT languages - such as Python, Java etc. - provide mechanisms to import functionalities into a running interpreter instance through the use of specific statements (e.g. import and from for Python). These statements allow embedding new functionalities at run-time, and therefore they are the most appropriate and easier mechanisms to be used in order to obtain the dynamic plug-in/-out of code in the RINA stack.

However, the user-space parts of the RINA stack are written in C++, thus the gap between the librina base language and the targeted interpreted language has to be filled by means of a target language binding interface. The binding interfaces are usually written in C, and this means that two transformations - librina (C++) to C, and C to the target language - need to take place in our case. The situation gets even worse if more target languages have to be supported, depending on the requirements of WP3, WP4 and WP5. Following this approach the situation would become problematic and difficult to manage.

To overcome the aforementioned problems, an automatic wrapping tool such as the Software Wrapper and Interface Generator (SWIG) is necessary. Refer to the [swig] section of this deliverable (7.2: Automatic software wrapping using SWIG) for further information.

## 4.2. Kernel-space plug-ins

Kernel-space plug-ins can be implemented through the use of loadable kernel modules (LKMs). LKMs are a simple way to extend the functionality of the kernel-space part of the stack, where the plugging/unplugging functionality is provided by the kernel modules subsystem.

A loadable kernel module (or LKM) is an object file that contains code to extend the running kernel, or so called base-kernel. LKMs are typically used to add support for new hardware, filesystems and functionalities in general. When the functionality provided by a LKM is no longer required, it can be unloaded in order to free memory and other resources.

Without loadable kernel modules, an operating system would have to include all possible anticipated functionality already compiled directly into the base kernel. Much of that functionality would reside in memory without being used, wasting memory, and would require that users rebuild and reboot the base kernel every time they require new functionality.

One minor criticism of preferring a modular kernel over a static kernel is the so-called "Fragmentation Penalty". The base kernel is always unpacked into real contiguous memory by its setup routines, so that the base kernel code is never fragmented. Once the system is in a state where modules may be inserted — for example, once the filesystems containing the modules have been mounted — it is probable that any new kernel code insertion will cause the kernel to become fragmented, thereby introducing a minor performance penalty.

For more information about the tools used to handle LKMs from command line, refer to [kmod].

# 5. High level architecture

This section describes the High Level Architecture of the RINA Plugin Infrastructure (RPI), which constitutes the core component of the PRISTINE SDK.

## 5.1. The RINA Plugin infrastructure

The following sections describe the RINA Plugin Infrastructure (RPI), used by plug-ins to override default configurable behaviours (policies) of the IRATI stack, making the stack programmable.

The RPI is intended to be an high level reference model that it is inherited by both user-space and kernel-space plugin infrastructures.

### 5.1.1. Overriding components' policies

The IRATI stack consists of many components - such as DTP, RMT or PFT - interacting with each other. Components reside in kernel-space, user-space or both. Some components, may be instantiated multiple times.

The RPI core concept is that each stack component may have a number of customizable functionalities referred to as policies. Policy come in two different flavours: (1) configuration parameters and (2) behaviours.

A parameter is a policy that can be assigned some value, - e.g. an integer or a string - so that the component uses that value during its operation. An example of parameter is the length of an RMT queue.

A behaviour is a policy that can be assigned some piece of code - e.g. a function, or a class method - to be executed when a certain event - the trigger - happens. As an example, the DTCP has the algorithm to use for Round Trip Time (RTT) estimation as a policy. This means that each time a DTCP instance needs to update the RTT estimation, the assigned policy is invoked to perform the computation.

The policies associated to each component are defined by the component specification. Each policy is required to have a default - e.g. a default value or a default behaviour - that is automatically used by the stack unless it is asked differently.

This said, the main purpose of RPI is to allow users to override default policies by means of plug-ins. A plug-in publishes custom policies to the stack, so that the users can dynamically select and replace them depending on their needs.

### 5.1.2. Policy set classes

A policy set is the atomic unit that the RPI can load and instantiate. It can be defined as a group of policies that the RPI system can load and instantiate without reaching an inconsistent state. This means a policy set for a specific component may eventually contain a single policy, if this does not interfere or cooperate with any other.

For the sake of having an easier initial prototype, however, it is convenient to wrap up policies in bigger policy sets, one for each IPCP component. In this simplified policy set model, as an example, the DTCP policy set groups together all the policies defined by the DTCP specification.

Policy sets are a convenient concept, since:

1. different behaviour policies in the same component may want to cooperate in order to achieve a common goal or interfere on shared resources
2. a plug-in can publish (see below) a whole policy set with a single RPI operation, instead of publishing the single policies individually

The policy sets are available in the local policy set catalogue (local-PSC). During the instantiation of an IPCP, and the components that an IPCP is made of, a policy set gets selected for each policy set scope. If no specific policy set is selected for a certain scope, the default one shall be used. Therefore, the stack allows to instantiate different policy sets (different set of behaviours) on different instances of a component. As an example, two different EFCP flows in the same IPCP process (maybe belonging to the same RINA application) corresponding to two different DTCP instances can have two different policy sets.

The following pseudocode shows an example of a class defining a policy set with three behaviour policies and two parameter policies:

```
class ExamplePolicySet {
    policyB1() : Integer
    policyB2(Integer: a, String: b) : Integer
    policyB3(String: g) : String

    policy P1: String
    policy P2: Integer
}
```

In addition to parametric policies, a specific policy set - e.g. a class inherting from ExamplePolicySet class - may define further tunable parameters, referred to as *policy-set-specific* parameters.

The latter parameters are opaque to the stack, since they are only used within the policy code. These parameters are exposed to the management user (see below) so that they can be used to tune the inner working of a policy set.

### 5.1.3. The RINA Plugin Infrastructure

By means of the RPI, a plug-in can publish and unpublish one or many policy sets.

A publish operation is used to make a policy set available to the local-PSC. A publish invokation must specify the type of policy set that implements (e.g. DTP), and a name used to identify the policy set itself.

Publishing must not be confused with selection. Once a policy set is available in the local-PSC, it may be selected - and so instantiated - during the instantiation of a new IPCP or during an IPCP policy run-time reconfiguration. The latter case may require appropriate locking techniques (e.g. Read-Copy-Update, or RCU), since a policy-set instance may be in use while a reconfiguration request comes.

The policy set selection procedure is reported in detail in section Section 5.2, “Policy-set selection”.

An unpublish operation is used to unregister a policy set from the local-PSC, so that the set will not be available to the stack anymore.

For the stack operation to be safe, a plug-in can be unplugged only when no policy-set instance managed by the plug-in is currently being used by the stack.

### 5.1.4. Plugin interaction with the component data model

The code implementing behavioural policies may need to access the data model of the component is acting on. As an example, an overrun policy for an RMT queue may want to carry out some operations on the queue itself, which are part of the RMT data model.

As a consequence, the plugin infrastructure must allow the running policy set instances to access the various data models. This can be done in different ways:

- Allow the policy code to directly access the data model of the components. From a low level perspective, the policy code would have a pointer or a reference to the data structure which constitutes the involved data model. This option gives maximum

flexibility to the policy implementation and it is straightforward to implement from the SDK point of view. The drawback, however, is that the stack is exposed to uncontrolled access to critical data structures. This may be acceptable for some high-performance policies. This approach is however hardly affordable in the IRATI prototype, since all the component's data model definitions are hidden inside the component implementation, and the component functionalities are exported through a per-component API.

- Allow the plugin to access the data model of the components only thorugh an ad-hoc API. The API would be different for each component type - e.g. an API for the DTCP, an API for the Flow Allocator - and would constrain the access to the data model in different ways depending on the specific component. The advantage of this approach is that critical data structures can be conveniently protected from uncontrolled access, and only the necessary functionalities are exposed. The drawback is on the implementation cost on the SDK size, since each component type will require an ad-hoc API to be used by the plugins.
- Allow the policy code to access a partial, specially tailored, copy of the internal state data-model. The stack, before calling the policy, would clone the necessary internal state of the component and expose the copy to the plugin. The advantage of this approach is that many critical data structures can be conveniently protected from uncontrolled access. The drawback is the overcost of cloning the internal data state, so this approach is only viable for non-performance critical policies.

## 5.2. Policy-set selection

This section describes how published policy sets are selected for being used by the stack.

### 5.2.1. Default policy sets

When a kernel-space or user-space plug-in publishes a policy set through the RPI interface (see Section 5.1.1, "Overriding components' policies"), that policy set becomes available to the user, and can be identified by a name provided by the plug-in at publication time.

The user - e.g. an application or the network administrator - can ask the stack for all the policy sets available for a certain component. At least one policy set, called default, must be available for each component. The default sets are built-in into the stack (e.g. it is published by an internal plug-in), and contain a callback function for each hook of the components on which they are defined. The default callbacks should be chosen to be as general purpose as possible, in order to be a good choice for most of the use cases.

When specific needs arise, so that the default callbacks for a certain component appear inappropriate, the user can use a plug-in to override part or all of them.

### 5.2.2. Identifying the components

In order for the user to select a policy-set for a running component there must be a way to name/identify the latter entity. All the policies - behavioural or parametric - apply to some components or sub-components of an IPC process. Therefore, the first thing that has to be specified is the identifier of the IPC process which contains the component to be selected.

An IPC process is made up of components that contain other components, and therefore can be represented with a hierarchical structure - a tree. The root of the tree represents the IPC process itself as a whole. The root's children represent the first level components of the IPC process (e.g. RMT, FA). Deeper levels of the tree represent sub-components or policies inside a (sub)component. Note that within this naming model a policy-set is identified as a child of the (sub)component it is associated to, and therefore can also be seen as a component.

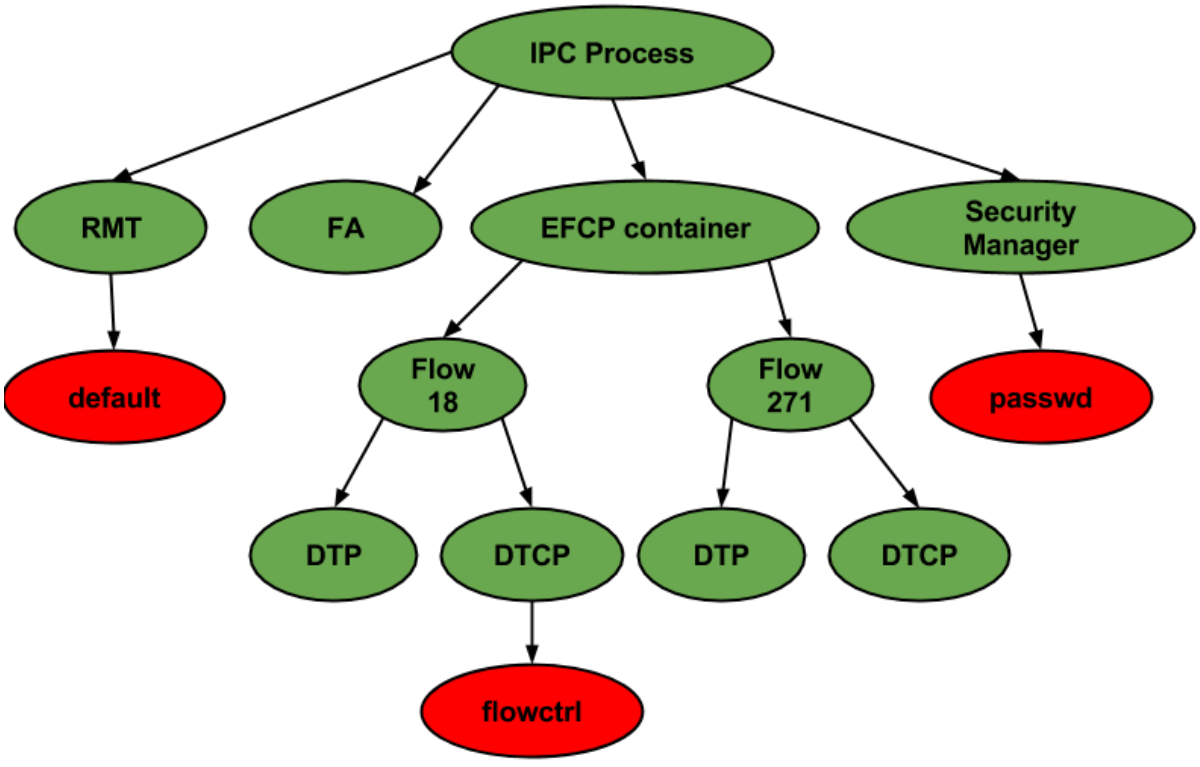


**Figure 6. IPC process component addressing example. Normal IPCP components are colored in green, while policy sets are colored in red.**

As a consequence of this tree structure, a component (possibly a policy-set) in the scope of an IPC process can be identified by the path that connects the root of the three to the component itself. As an example, the DTCP instance for flow 18 of the IPC process (partial) tree shown in the figure is identified by the path EFCP container --> Flow 18 --> DTCP. Using a dot notation, the involved DTP component may be identified by the string efcp.18.dtcp.

### 5.2.3. Policy set selection

When an user wants to apply a policy to a running component, it instructs the the IPC Manager daemon (e.g. via console) to perform a select-policy-set operation, where the following arguments must be specified:

- The identifier of the IPC process where the involved component resides
- The identifier of the component, as specified in section Section 5.2.2, “Identifying the components”.
- The name of the policy to be applied. This is the same name published by the plug-in (e.g. a kernel module or a shared libray) itself when using publish operation (see Section 5.1.1, “Overriding components' policies”).

As an example, the user may issue the following request to select the policy-set published as "foo-policy-set" for a DTCP instance

```
select-policy-set 16 efcp.2.dtcp foo-policy-set
```

In order to deliver the select-policy-set request to the right IPC process component, the IPC Manager will trigger a Component Configuration Delivery Workflow, as described in the Section 5.2.5, “Component Configuration Delivery Workflow” section.

### 5.2.4. Setting tunable parameters

As described in Section 5.1.1, “Overriding components' policies”, a policy set may have two kinds of tunable parameters

- parametric policies, i.e. policies that are parameters, visible to the stack since they are part of an IPC process component
- policy-set-specific parameters, i.e. tunable parameters associated to a specific policy-set, are not visible to the stack since they are not part of an IPC process component

The user can instruct the IPC manager to issue a set-policy-set-param configuration request in order to modify these parameters. The following arguments have to be specified:

- The identifier of the IPC process where the involved component resides
- The identifier of the component to address, as specified in section Section 5.2.2, "Identifying the components".
- The name of the parameter to be set
- The value to set

As an example, the user may set an RMT parametric policy through the following request:

```
set-policy-set-param 16 RMT MaxQueueLen 280
```

As another example, the user may set a policy-set-specific parameter for the *passwd* Security Manager policy-set through the following request:

```
set-policy-set-param 4 security-manager.passwd MaxRetries 8
```

In order to deliver the set-policy-set-param request to the right IPC process component (possibly a policy-set), the IPC Manager triggers a Component Configuration Delivery Workflow, as described in the Section 5.2.5, "Component Configuration Delivery Workflow" section.

### 5.2.5. Component Configuration Delivery Workflow

This section illustrates how a policy-related configuration request travels through the stack and is delivered to the IPC process component addressed by the request itself. The mechanisms presented here are used to serve both the select-policy-set and the set-policy-set-param requests.

When the IPC Manager receives one of these two commands from the built-in console, it uses the IPC process identifier argument to find the right IPC process daemon to which the request is going to be forwarded. In the IRATI prototype, the request is sent through a netlink message, using the RNL infrastructure.

Upon receiving the request message, an IPC process daemon uses the component identifier string to decide to which component (e.g. Flow Allocator Task, Enrollment Task) it should forward the request to. As described in Section 5.2.2, "Identifying the

components" the component identifier is a string composed of a list of substrings separated by dots ("."). The IPC process daemon will examine only the first substring to see if it matches the name of some its component. If there is no match, the request is invalid, and an error message is returned to the IPC Manager. If there is a match, the IPC process daemon will remove the first substring to the list and forward the request to the matched component, which will receive the updated identifier string.

The procedure above for an IPC process is actually a special case of a more general recursive procedure carried out by any component to forward, accept or deny a configuration request:

1. If the first substring in the component identifier string matches the name of some sub-component of the current component, remove the substring from the identifier and forward the (modified) request to the matching sub-component. The sub-component may be a policy set.
2. If the component identifier is empty, it means that the current component is the the destination of the configuration request. From this point on the actions to be taken depends on whether the request is a select-policy-set or a set-policy-set-param.
   - For a set-policy-set-param request, examine the name of the parameter to be configured. If it matches some tunable parameter for that component, then carry out all the operations needed to set the matching parameter to the argument value, and notify the success to the parent component. If the parameter name does not match any tunable parameter for the current component, or if the value argument is invalid, notify the error to the parent component.
   - For a select-policy-set request, first of all make sure that the current component is not a policy-set - it would not make any sense to apply a policy-set to another policy-set - notifying the error to the parent component if the case. Then examine the name of the policy-set to be applied. If it matches some policy-set published for that component, then carry out all the operations needed to select the matching policy-set for the current component, and notify the success to the parent component. If the specified policy-set name does not match any published name for the current component, notify the error to the parent component.
3. If no sub-component matches the first substring, then notify the error to the parent component.

Once the request has been processed, either in the kernel or in the user-space IPC process, a response message is produced to be delivered to the IPC Manager.

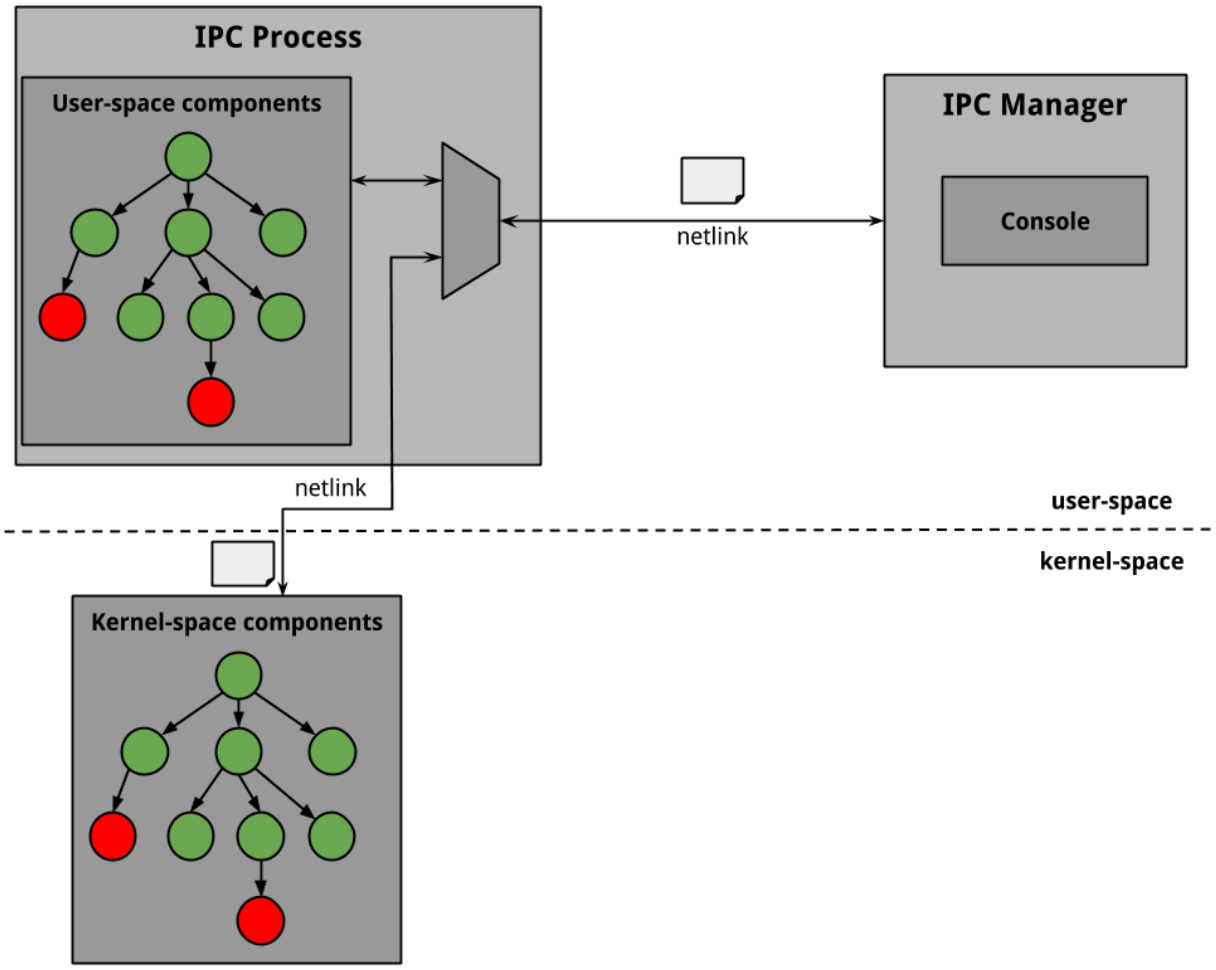


**Figure 7. The configuration delivery workflow**

## 5.3. The Kernel space RINA Plugin Infrastructure

This section describes the kernel-space RINA Plugin Infrastructure (kRPI). In spite of the general definition of policy sets, the initial kernel-space SDK prototype will assume the simplified one-policy-set-per-component model, as explained in Section 5.1.1, “Overriding components' policies”

### 5.3.1. Kernel-space hooks

Functionality plugging is possible in the stack by means of hooks. An hook is a location in the code where the stack invokes a function - the hook callback - without knowing its implementation. This can be achieved through an indirect function call.

Hooks are usually placed in code locations that correspond to some event happening - e.g. a PDU is dropped because of a queue is full, a retransmission timer fires, a recomputation/update of a Round Trip Time (RTT) has to be performed.

The role of an hook callback, in general, is to respond to an event by executing some action and/or returning some information to the stack. An hook, therefore, is a place where a policy can be applied, i.e. there is a one-to-one relationship between policies and hooks.

### 5.3.2. kRPI policy sets

A kRPI policy set is a data structure used by kRPI to represent the policy set concept introduced in sec. Section 5.1.1, “Overriding components' policies”.

This data structure is essentially a container of function pointers, each one corresponding to a different hook of the same component. A different data structure exists for each component, since components have different hooks - e.g. the callback have different prototypes. Apart from function pointers, a kRPI policy set data structure contains

- a reference to the component Data Model (DM), so that the hook callbacks can access the DM of the component instance they are working on
- the parameter policies for the specific component and
- a pointer to a per-instance data structure that stores a) one or more policy-set-specific parameters that can be tuned by user-space management entities and b) other private data not accessible outside the plugin.

An abstract example of kRPI policy set for the component 'Example' is the following:

```
struct ExamplePolicySet {
    void    (*policy1)(ExamplePolicySet *ps, int, unsigned);
    int     (*policy2)(ExamplePolicySet *ps);
    char    *(*policy3)(ExamplePolicySet *ps, unsigned, unsigned);

    int    param1;
    char   *param2

    ExampleDM         *data_model;
    void              *private_data;
};
```

The policy-set-specific parameters are used by the callback functions to perform their tasks and can be tuned by user applications or the network administrator. They must not be confused with parameter policies - e.g. policies that are parameters - such as param1 in the example. Different policy sets defined on the same policy set

class may have different specific parameters, even though they are defined on the same hooks set. As an example, two different policies that performs RTT estimation - corresponding to the same hook - may have different policy-specific-parameters, depending on the estimation algorithm employed. This implies that policy-set-specific parameters cannot be directly used by the stack, but they are private to the policy set callbacks. The parameter policies can be read from both the stack and the callbacks.

The hook callbacks, therefore, may access four kinds of data:

- non-tunable private data (private_data)
- policy-set-specific parameter values (also private_data)
- parameter policies (param1 and param2)
- the DM of the instance the callbacks are operating on (data_model).

### 5.3.3. kRPI interface

Kernel-space plug-ins can be implemented through the use of loadable kernel modules (see Section 4.2, “Kernel-space plug-ins”), so that an existing dynamic loading mechanism is reused.

As an example, a LIFO scheduler policy-set for RMT could be loaded with the following shell command:

```
# insmod rmt-lifo.ko
```

A kernel module that wants to plug-in functionalities in the kernel-space stack can publish and unpublish policy sets using the kRPI API, accordingly to what described in Section 5.1.1, “Overriding components' policies” Publishing is normally expected to happen in the module init function - even though this is not a requirement. Similarly, unpublishing is expected to happen in the module exit function.

#### Policy-set factories

Policy sets are published using *policy-set factories*. Each factory contains:

- The name of the policy-set, to be used as an argument of the IPC Manager console select-policy-set command.
- A constructor method to create policy-set instances. The constructor method should initialize the policy-set instance filling in the policies - e.g. pointers to callbacks

implemented by the plugin - and allocating the private data to be used by the callbacks.

- A destructor method to destroy policy-set instances.

The following definition is used for kernel-space policy-set factories:

```
struct ps_factory {
    /* A name for this policy-set. */
    char name[POLICY_SET_NAME_MAX_LEN];

    /* Constructor method. */
    struct ps_base * (*create)(struct rina_component * component);

    /* Destructor method. */
    void (*destroy)(struct ps_base *);
};
```

Each kernel-space component expose two API functions to let plugins publish and unpublish policy-set factories. The API for RMT is the following:

```
int rmt_ps_publish(struct ps_factory *factory);
int rmt_ps_unpublish(const char *name); /* 'name' is the name of a policy-
set */
```

The APIs for the other components will be similar - i.e. have the same function signatures.

## Policy-set lifecycle

When an instance of kernel-space component X is instructed to select the policy set Y, the component will use the Y's factory create method to instantiate a new policy set instance, and bind that new instance to the X instance. Until the X instance is asked to change again its policy set, the stack code for X will use the policies specified by Y whenever they are needed.

When the X instance is going to be destroyed or is asked to select a different policy-set, the Y's factory destroy method is used to destroy the Y instance bound to the X instance.

The following figure contains a simple UML sequence diagram that depicts the lifecyle of a policy set instance for the RMT component.

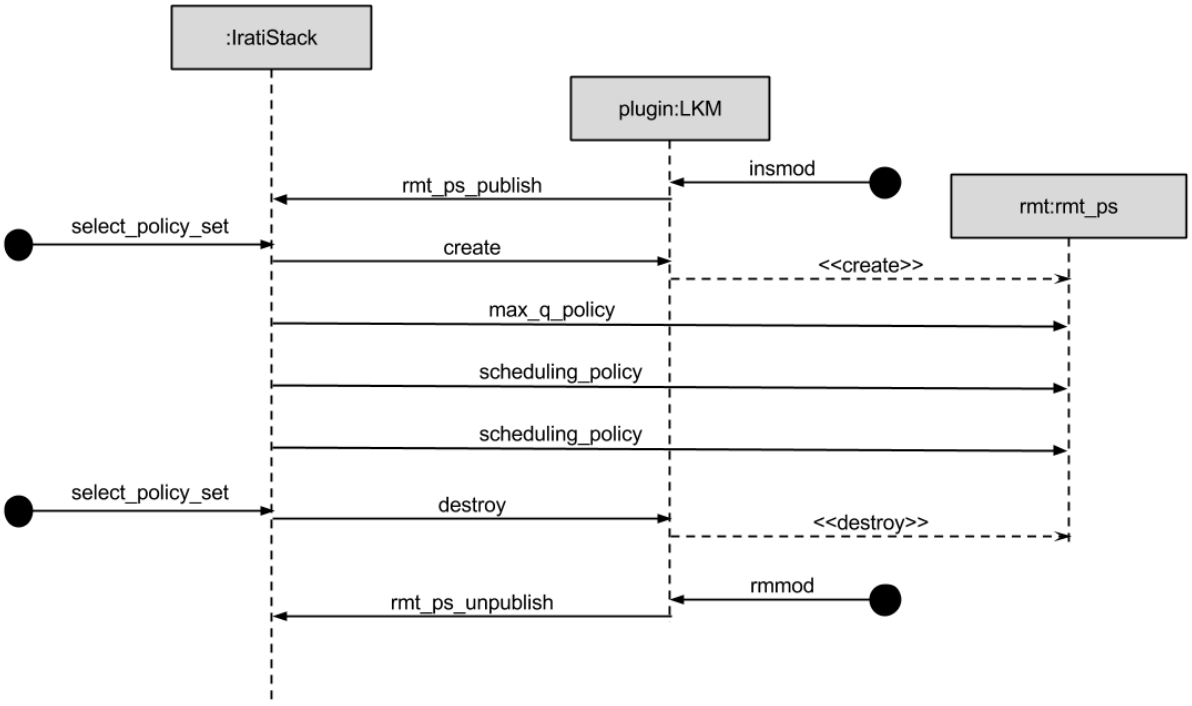


**Figure 8. Lifecycle of a policy set instance for RMT**

## Policy-set classes

For the sake of a good software design - and also to enforce some compile-time type checking - kernel-space policy-set classes are defined using an Object Oriented approach:

- All policy-set classes derive from a common base class - struct ps_base
- All components derive from a common base class - struct rina_component. For the time being this class is empty, and its only purpose is to enforce static type checks at compile time.

Inheritance can be implemented in C by containment, since the language guarantees that the address of a struct is the same as the address of its the first member

```
struct Base {
    // ...
};

struct Derived {
    struct Base b;  /* Base class object must be the first member. */
    // ...
};
```

Containment is not the only way to implement inheritance in C, but it has been chosen because it is a valid solution to the problem and is also straightforward to implement.

Arguments and return values in the policy-set factory methods are pointers to base classes, but the plugin must be implemented to operate on the derived classes, performing upcast and downcast operations where appropriate.

As a concrete example for the RMT policy-set factory:

- The stack will invoke the create method passing a pointer to a struct rina_component object that is actually contained inside a struct rmt object. The plugin can safely downcast it to a struct rmt pointer.
- The create method creates a struct rmt_ps object but returns an upcasted pointer to struct base_ps to the stack.
- The stack will invoke the destroy method passing a pointer to a struct ps_base object that is actually contained inside a struct rmt_ps object. The plugin can therefore safely downcast it to a struct rmt_ps pointer.

The following code contains definitions of the classes involved in the examples

```
struct ps_base {
    /* Method for setting policy-set-specific parameters. */
    int (*set_policy_set_param)(struct ps_base * ps, const char *
 name, const char * value);
};

struct rmt_ps {
    struct ps_base base;
    /* Behavioural policies. */
    void (* max_q_policy_tx)(struct rmt_ps *, struct pdu *, struct rfifo
 *);
    void (* max_q_policy_rx)(struct rmt_ps *, struct sdu *, struct rfifo
 *);
    /* Parametric policies. */
    int max_q;
    /* Reference used to access the RMT data model. */
    struct rmt * dm;
    /* Data private to the policy-set implementation. */
    void *priv;
};
```

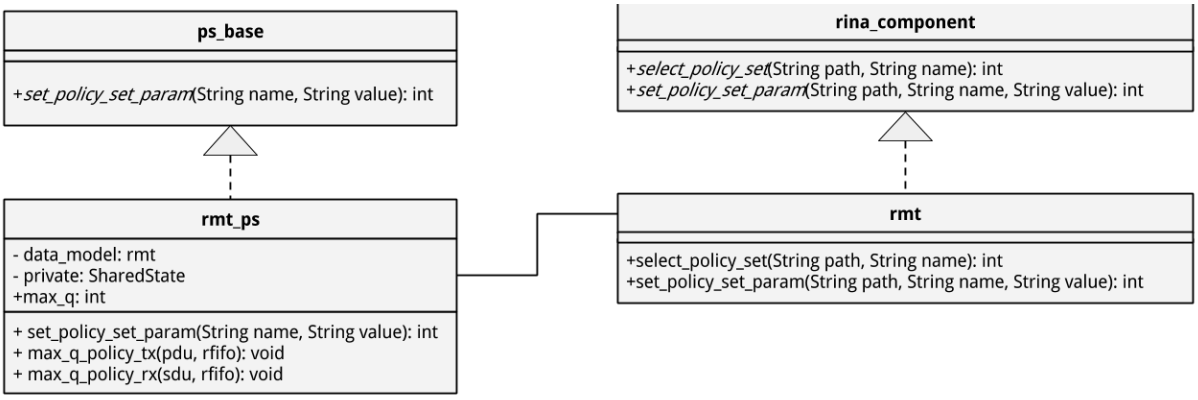


**Figure 9. UML class diagrams for RMT components and policy-sets**

### Plugin unloading

Plugins can be unloaded removing the corresponding kernel module

```
# rmmod rmt-lifo
```

However, a kernel plugin module cannot be unloaded until all the existing policy-set instances created by that plugin have been destroyed, otherwise invalid memory accesses will occur as soon as a callback is invoked. For this reason, the kRPI framework implements a reference counting mechanism that prevents a module to be unloaded while there are alive policy sets objects belonging to that plugin. This is achieved simply incrementing the module reference counter each time a policy-set (published by that module) is created and decrementing the reference counter each time a policy-set is destroyed.

## 5.4. The User space RINA Plugin Infrastructure

This section describes the user-space RINA Plugin Infrastructure (uRPI). In spite of the general definition of policy sets, the initial user-space SDK prototype will assume the simplified one-policy-set-per-component model, as explained in Section 5.1.1, “Overriding components' policies”

### 5.4.1. Policy sets and inheritance

A uRPI policy set class is a data type used to represent the policy set concept introduced in Section 5.1.1, “Overriding components' policies”

An uRPI policy sets class for a certain component contains a method for each behavioural policy of the component. A different class exists for each component, since

each component has its own policies. Apart from policy methods, a policy set class contains:

- a data member for each parameter policy
- a reference to the Data Model (DM) of the component, so that the methods can access the DM of the component instance they are working on.

An abstract example of the uRPI policy set class for the (fake) component 'Example' is the following

```
class ExamplePolicySet {
    public:
        ExamplePolicySet(DataModel *dm);

        // Policies which are behaviours
        virtual void  policy1(int, unsigned) = 0;
        virtual int   policy2() const = 0;
        virtual char* policy3(unsigned, unsigned) = 0;

        // Policies which are parameters
        int policy4;
        float policy5;

    protected:
        ExampleDataModel *data_model;
};
```

A convenient way to plug-in custom policy sets in the user-space part of the IRATI stack is by making use of two Object Oriented (OO) programming techniques called inheritance and polymorphism. Since the user-space stack is written in C++, which supports Object Oriented programming, this choice is actually possible.

Through inheritance, an existing class can be extended to add new methods and objects, or to override existing methods.

The ExamplePolicySet class is intended to be abstract - i.e. a class that cannot be instantiated directly. Its methods - the behavioural policies - are actually implemented in one or more derived classes.

A plug-in that wants to define a custom policy set for the 'Example' component, can extend the ExamplPolicySet class to override the policy members, as illustrated by the following code:

```
class CustomExamplePolicySet : public ExamplePolicySet {
    public:
        CustomExamplePolicySet(DataModel *dm);

    private:
        // Private data to be used by the overridden policies
        int a;
        // ...
};

void CustomExamplePolicySet::policy1(int, unsigned)
{
    // ...
}

int CustomExamplePolicySet::policy2() const
{
    // ...
}

char* CustomExamplePolicySet::policy3(unsigned, unsigned)
{
    // ...
}
```

As the example shows, the CustomExamplePolicySet overrides (implement) the behavioural policies.

When the stack needs to invoke a policy on an instance of 'Example', it uses a reference (pointer) to an ExamplePolicySet object, which can only refer to an object belonging to some derived class (e.g. CustomExamplePolicySet object). Because of polymorphism, different policies are invoked, depending on the actual type of the object being referenced. In this way the user-space stack code can transparently use policies, without knowing what policy set is actually being used on a certain component instance.

### 5.4.2. Dynamic loading of C++ classes

According to what explained in the previous section, a plug-in can provide custom policy sets for a certain component by implementing a subclass of a policy set abstract class.

In order to make the subclass code available to the stack, however, it is necessary to dynamically load that code, e.g. load it while the stack is running. This can be

accomplished using different methods and tools, reported in this section. All these methods assume that the plug-in is contained in a dynamically linkable library (e.g. a shared- object on Unix-like systems).

### libdl

On POSIX systems, libdl ([libdl-online]) is the basic C library used to perform dynamic loading of C code. The main functions exposed by libdl are dlopen and dlsym. The dlopen function is used to load a shared library given its name (a string). The dlsym function is used to extract a symbol from the loaded code, given its name (a string), so that the symbol - e.g. a function - can subsequently used.

However, libdl does not allow to directly load C++ classes, because libdl does not support C++ symbols, which are mangled by the C++ compilers (in non-standard ways).

Neverthless, a common workaround is illustrated in [libdl-cpp-workaround], and summarized here. Luckily, the workaround is built on the inheritance and polymorphism concepts, which is exactly what it is needed in our case. Dynamically loading of C++ code is possible with libdl, provided that the symbols to export are not mangled: This can be achieved prefixing the symbol definitions with the extern "C" qualifier. For each custom policy set that the plug-in wants to export a couple of extern "C"-qualified factory functions have to be exported to the stack: a constructor and a distructor, which operates on a pointer to the base class.

The two factory functions are imported into the stack using dlsym, as normal C functions. When the stack wants to instantiate a policy set exported by the plug-in, it will invoke the constructor function. Because of polymorphism, the stack can use the created object without knowing its actual type. When the policy set object is to be destroyed, similarly, the destructor method is invoked.

A practical example is shown in section Section 5.4.3, “uRPI interface”.

### Boost extension

The Boost libraries provide a cross-platform C++ library to perform dynamic loading of functions and classes, named boost::extension Essentially, the mechanisms are similar to the ones reported in the libdl section - and libdl is still used to implement it on Unix-like systems - but they are conveniently wrapped by a high level OO library.

It is possible to find documentation and tutorials about boost::extension at [boost-extension].

### 5.4.3. uRPI interface

For the initial SDK prototype, libdl has been chosen as a dynamic loading technology, since

1. libdl functionality are enough to solve the problem without much effort
2. we avoid to introduce a dependency towards the Boost libraries.

#### Plugin loading

The IPC process userspace daemon is in charge of using libdl to load the plugins when needed. In order to load and unload userspace plugins, the administrator can use the IPCM console.

Once the IPCP daemon has loaded a plugin, it uses dlsym to dynamically load the init routine that each plugin must define. Such init routine must have the following signature:

```
extern "C" int init(IPCProcess * ipc_process);
```

The IPCP daemon will refuse to load a plugin for which this routine is not defined. Once loaded the init routine is invoked, passing a reference to the IPC Process daemon main class as an argument. A plugin should use its init routine to publish all the policy sets it implements.

The IPCM implements the plugin-load and plugin-unload commands that can be used to load a plugin installed on the local filesystem into a specified IPC Process daemon.

These command accept two arguments:

1. The identificator of the IPC process that has to load the plugin (an unsigned integer in the IRATI stack)
2. The name of the plugin to be loaded.

In the following example

```
    IPCM >>> plugin-load 2 sm-passwd
```

the IPC process daemon 2 will look for a shared object called 'sm-passwd.so' in the local filesystem, and load it if found.

In this other example:

```
IPCM >>> plugin-unload 2 sm-passwd
```

the IPC process daemon 2 will unload the sm-passwd plugin (if previously loaded).

## Policy-set factories

Policy sets are published using *policy-set factories*. Each factory contains:

- The name of the policy-set, to be used as an argument of the select-policy-set command.
- The component to which the policy set applies to.
- A constructor method to create policy-set instances. The constructor method should create an instance of a policy-set class that implements the behavioural policies, as explained in the next section.
- A destructor method to destroy policy-set instances.

The following definition is used for user-space policy-set factories:

```
struct PsFactory {
      // Name of this pluggable policy set.
      std::string name;

      // Name of the component where this plugin applies.
      std::string component;

      // Constructor method for instances of a pluggable policy set.
        extern "C" IPolicySet *(*create)(IPCProcessComponent *ctx);

      // Destructor method for instances of a pluggable policy set.
      extern "C" void (*destroy)(IPolicySet *ps);
};
```

Publishing and unpublishing is possible by means of the following API exposed by the IPC Process daemon main class:

```
int IPCProcess::psFactoryPublish(PsFactory fact);
int IPCProcess::psFactoryUnpublish(std::string component, std::string
 name);
```

A reference to the IPC Process daemon main class instance - necessary to use the API - is available in the init function, as explained in the previous section.

## Policy-set lifecycle

When an instance of user-space component X is instructed to select the policy set Y, the component will use the Y's factory create method to instantiate a new policy set instance, and bind that new instance to the X instance. Until the X instance is asked to change again its policy set, the stack code for X will use the policies specified by Y whenever they are needed.

When the X instance is going to be destroyed or is asked to select a different policy-set, the Y's factory destroy method is used to destroy the Y instance bound to the X instance.

The following figure contains a simple UML sequence diagram that depicts the lifecyle of a policy set instance for the Security Manager component.

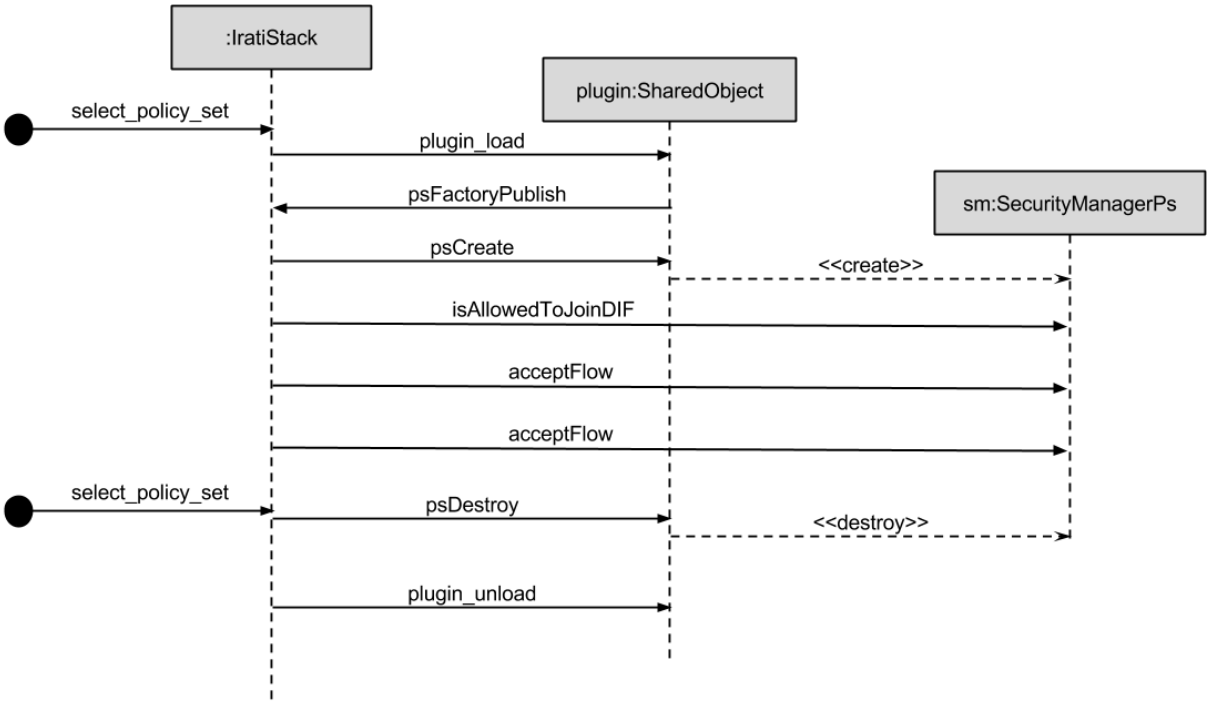


**Figure 10. Lifecycle of a policy set instance for Security Manager**

## Policy-set classes

As explained in section Section 5.4.1, "Policy sets and inheritance" user-space policy-set classes constitute a class hierarchy. The hierarchy has three levels:

1. A common abstract base class - class IPolicySet - from which all the policy-set classes derive from. This class contains the interface common to all policy-set classes.
2. An abstract class for each component, deriving from class IPolicySet, and from which all the policy-set classes for the same component derive from (e.g. class

ISecurityManagerPs for Security Manager). The second-level classes define the behavioural policies - as function member - and parametric policies - as data members - that apply to each IPCProcess component.

3. The concrete policy-set classes implemented by plugins. These classes derive from the second-level class corresponding to the component they apply to.

A three-levels class hierarchy exists also for IPC Process components:

1. A common abstract base class class IPCProcessComponent (already present in the IRATI prototype), from which all the component classes derive from. This class contains the interface common to all IPC Process components.
2. An abstract class for each component, deriving from class IPCProcessComponent, and from which the corresponding component class - part of the IPC Process deamon implementation - derives from. The second-level classes define the interface of the IPC Process components, that can be used by the plugins.
3. A concrete class for each IPCProcess component, deriving from the corresponding second-level abstract class, that provides the actual implementation.

An extract of these three-levels hierarchies is shown in the following figure, showing the portion of the hierarchy related to the Security Manager.

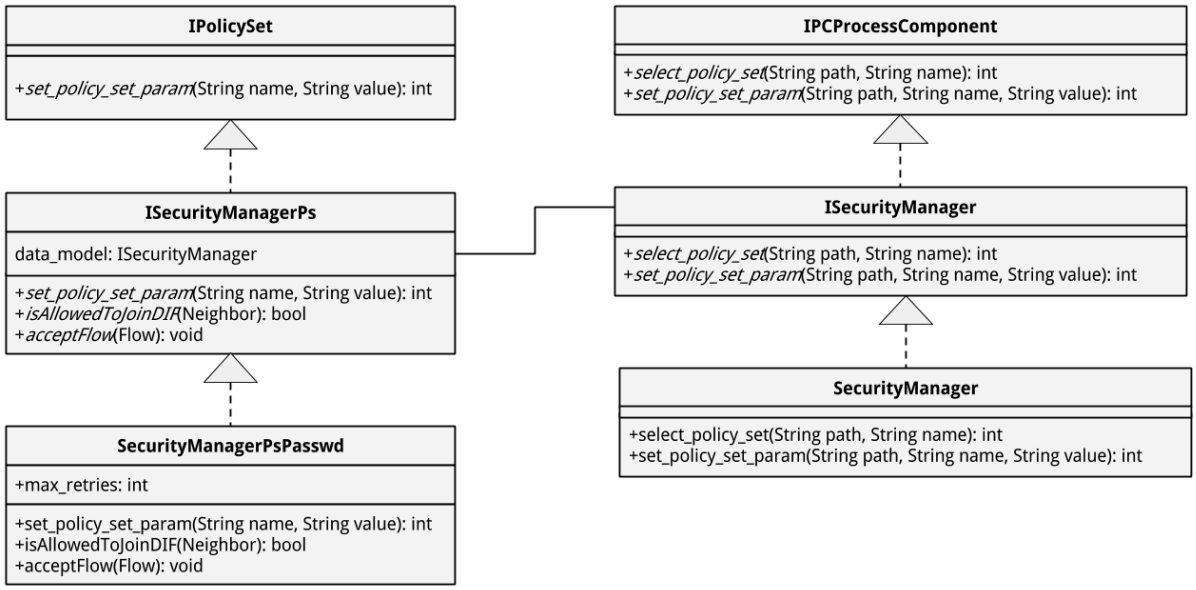


**Figure 11. UML class diagrams for Security Manager component and policy-sets**

The need for the second hierarchy level comes from the fact that plugin's code is loaded dynamically:

- In order for the IPC Process daemon to invoke behavioural policies (implemented inside the plugin), an abstract class with pure virtual functions only is needed. This role is accomplished by the second-level in the policy-set classes hierarchy - e.g. ISecurityManagerPs.
- In order for a concrete policy-set class (implemented inside a plugin) to invoke the functionality exposed by the associated component data model, an abstract class with pure virtual functions only is needed. This role is accomplished by the second-level in the IPC Process component classes hierarchy - e.g. ISecurityManager.

Arguments and return values in the policy-set factory methods are pointers to base classes, but the plugin must be implemented to operate on second or third level derived classes, performing upcast and downcast operations where appropriate.

As a concrete example for a Security Manager policy-set factory that manages specific SecurityManagerPsPasswd concrete policy-set class:

- The stack will invoke the create method passing a pointer to a class IPCProcessComponent object that actually points to a class SecurityManager object. The plugin can safely downcast it to a ISecurityManager pointer.
- The create method creates a class SecurityManagerPsPasswd object but returns an upcasted pointer to class IPolicySet to the stack. The stack can safely downcast it to an ISecurityManager object.
- The stack will invoke the destroy method passing a pointer to a class IPolicySet object that actually points to a class SecurityManagerPsPasswd object. The plugin can therefore safely downcast it to a class SecurityManagerPsPasswd pointer.

The following code contains the definitions of the classes involved in the previous examples

```
class IPolicySet {
public:
    /* Method for setting policy-set-specific parameters. */
    virtual int set_policy_set_param(const std::string& name,
                                     const std::string& value) = 0;
    virtual ~IPolicySet() {}
};

class ISecurityManagerPs : public IPolicySet {
// This class is used by the IPCP to access the plugin functionalities
public:
        /// Decide if an IPC Process is allowed to join a DIF
```

```
        virtual bool isAllowedToJoinDIF(const rina::Neighbor& newMember)
 = 0;

        /// Decide if a new flow to the IPC process should be accepted
        virtual bool acceptFlow(const Flow& newFlow) = 0;

        virtual ~ISecurityManagerPs() {}
};

class ISecurityManager: public IPCProcessComponent {
// This class is used by the plugins to access the IPCP functionalities
public:
        virtual ~ISecurityManager() {}
};

class SecurityManager: public ISecurityManager {
// Used by IPCP to access the functionalities of the security manager
private:
        IPCProcess *ipcp;
public:
        ISecurityManagerPs * ps;

        SecurityManager();
        void set_ipc_process(IPCProcess * ipc_process);
        void set_dif_configuration(const rina::DIFConfiguration&
 dif_configuration);
        int select_policy_set(const std::string& path, const std::string&
 name);
        int set_policy_set_param(const std::string& path,
                                 const std::string& name,
                                 const std::string& value);
        ~SecurityManager() {};
};

class SecurityManagerPs: public ISecurityManagerPs {
public:
        SecurityManagerPs(ISecurityManager * dm);
        bool isAllowedToJoinDIF(const rina::Neighbor& newMember);
        bool acceptFlow(const Flow& newFlow);
        int set_policy_set_param(const std::string& name,
                                 const std::string& value);
        virtual ~SecurityManagerPs() {}

private:
        // Data model of the security manager component.
        ISecurityManager * dm;
```

```
};
```

# 6. Bindings for high-level programming languages

In a general sense, language bindings are Application Programming Interfaces (API) interfacing a programming language to a library or operating system service. Bindings provide the "glue code", i.e. the code that is needed in between both parts, to use the interfaced libraries or services from a certain programming language. This "glue code" comes in the form of wrapper libraries to bridge the two programming languages. Using these wrapper libraries, a program written in a certain language is able to call or use a library written in another language.

The major benefits of language bindings are software reuse and code efficiency. Reusing software allows to use already existing libraries without the need of implementing the library in several languages. In the same way, system programming languages such as C or C++ are used for implementing efficient-critic functionalities, which is sometimes impossible to reproduce in high-level languages such as Java. To that extent, wrapping libraries can be used to interface high-level languages to leverage the efficiency achieved by C/C++ code.

In PRISTINE, librina is implemented in C. In order to allow the implementation of RINA applications or policies in a different language than C, language bindings are needed. This section describes how these bindings are implemented in PRISTINE. The tool chosen to automatize the wrapper libraries' creation process is SWIG [swig]. In the following we explain the general process of obtaining bindings, the SWIG approach, the changes needed depending on the language and the "manual" process of creating bindings for Java and Python using SWIG.

## 6.1. High level language bindings

This section presents an overview of the general process for obtaining bindings for high-level languages, in order to illustrate the binding creation process to later focus on automatic binding creation using SWIG. We will focus on Java and Python since those are the ones considered for future implementations of RINA based applications and policies, eventually. Knowing how language bindings are created manually will help to understand the latter approaches using SWIG and how to use the wrapping libraries.

### 6.1.1. Java Native Interface

The Java Native Interface (JNI) [java-ni] is a programming framework that allows Java programs - running in a Java Virtual Machine (JVM) - to use/call and be used/called by native software.

The term "native" refers to dedicated software supported by a certain system which is specifically designed and implemented for it (involves minimal computational overhead). Those programs are usually written in system programming languages such as C, C++ or assembly.

When an application cannot be written entirely in Java (e.g. the standard Java API does not support the required functionality or the required functionality is already written in another language), JNI can be used to write native methods to handle these situations. In turn, many of the standard Java libraries use JNI to provide functionality such as I/O operations, graphic and sound capabilities, etc.

JNI allows native methods to use Java objects, which can be created by the native method itself or the Java application code. Then, the native method can inspect or use those created Java objects to perform the corresponding tasks.

## Calling C/C++ methods from Java code

Native methods are implemented in independent C or C++ files. When the Java program calls the native method, it passes a "JNIEnv" pointer, a "jobject" pointer and the arguments declared in the Java method. A native method may look like the following.

```
#include "ClassName.h"

JNIEXPORT void JNICALL Java_ClassName_MethodName
  (JNIEnv *env, jobject obj)
{
    /* Native method code */
}
```

The "env" pointer contains the interface to the JVM, which allows to access the necessary functions to interact with the JVM and to handle Java objects. The pointer "obj" refers to the Java object in which the native method has been declared. The include line refers to a C header file that contains functions with the correct signatures for all of the native methods defined in that class. These headers can be obtained by using the javah tool (provided by the JDK) using the source Java class as argument. In this case, ClassName.h will contain the following line:

```
JNIEXPORT void JNICALL Java_ClassName_MethodName(JNIEnv *, jobject);
```

We then must build the C code to get a dynamic library (e.g. .dll or .so depending on the platform) to be referenced from the Java code. To that extent, we must tell the C compiler where to find the JNI headers, which reside in the jdk/include directory of the JDK installation path. To that extent, we must include the jdk/include directory and any other directory present for the given platform.

The call of the native method from the Java code would look like this:

```
class ClassName{
 private native static void methodName();
 public static void main(String[] args) {
  methodName();
 }
 static{
  System.loadLibrary("LibraryName");
 }
}
```

Where the "native" modifier defines the signature of the method that is implemented natively, and "LibraryName" is the name of the dynamic library that contains the native method. The System.loadLibrary() call takes the file name of the library (such as .dll or .so files). Then, the native method can be called like any other Java method.

## Type mapping

Native C/C++ data types can be mapped to/from Java data types. Compound types (objects, arrays, strings, etc.) the native code must explicitly convert the data calling methods through JNIEnv. The following table shows the mapping of types between JNI (Java) and native code.

**Table 1. Type mapping between native code and Java**

| Native Type | Java Language Type | Description | Type signature |
|---|---|---|---|
| unsigned char | jboolean | unsigned 8 bits | Z |
| signed char | jbyte | signed 8 bits | B |
| unsigned short | jchar | unsigned 16 bits | C |
| short | jshort | signed 16 bits | S |
| long | jint | signed 32 bits | I |

| Native Type | Java Language Type | Description | Type signature |
|---|---|---|---|
| long<br>long__int64 | jlong | signed 64 bits | J |
| float | jfloat | 32 bits | F |
| double | jdouble | 64 bits | D |
| void | | | V |

## Language bindings with JNI

Language bindings in JNI are based on the concepts presented above. If we have a certain C code that we want to make accessible from Java, we must create the glue code to bind both programming languages by means of native methods. We have already seen how a C method can be called from Java code, so the necessary step to create the bindings is to make calls to the desired C methods from the implemented native methods. Therefore, our implemented native methods will form the glue code.

Let's have a look at a Hello World example to illustrate the complete process.

Assume that we have the following C code that we want to call from a Java program.

### helloWorld.h:

```
void print();
```

### helloWorld.c:

```
#include <stdio.h>

void print()
{
    printf("Hello World!\n");
    return;
}
```

The first step is to write the Java code that makes use of the method.

### HelloWorldJava.java

```
class HelloWorldJava {
 private native static void print();
```

```
  public static void main(String[] args) {
   print();
  }
  static{
   System.loadLibrary("libHelloWorld");
  }
}
```

And the glue code to bind both languages.

**libHelloWorld.c:**

```
#include <stdio.h>
#include "HelloWorld.h"
#include "HelloWorldJava.h"

 JNIEXPORT void JNICALL
 Java_HelloWorld_print(JNIEnv *env, jobject obj)
 {
     print();
 }
```

Then, we can create a convenience build script like the following.

**make.sh:**

```
#!/bin/sh

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
javac HelloWorldJava.java
javah HelloWorldJava
gcc -shared libHelloWorld.c -o libHelloWorld.so
java HelloWorld
```

And execute it.

```
$ chmod +x make.sh
$ ./make.sh
```

### 6.1.2. Python extensions

The Python "extension modules" can be used to add new built-in Python modules and call external C library functions or system calls.

In the following we will explain how to make C functions callable from Python. Assuming we have a C function "myFunction()" (that takes a string and returns an integer) in a C library "myLibrary", we want to make it callable from Phyton as follows:

```
>>> import myLib
>>> myInt = myLib.myFunction("myString")
```

## Calling C/C++ functions from Python

To support extensions, the Python API (Application Programming Interface) defines a set of functions, macros and variables that provide access to most aspects of the Python run-time system. To incorporate the Python API in a C source file the header "Python.h" must be included. All user-visible symbols defined by Python.h have a prefix of Py or PY, except those defined in standard header files. For convenience, and since they are used extensively by the Python interpreter, "Python.h" includes a few standard header files: <stdio.h>, <string.h>, <errno.h> and <stdlib.h>.

The following C module includes the myLib_myFunction() function, which is the C function that will be called when the Python expression myLib.myFunction("myString") is evaluated. We will explain later how it is actually called.

### myLibModule.c

```
#include <Python.h>

static PyObject *
myLib_myFunction(PyObject *self, PyObject *args)
{
    const char *myString;
    int myInt;

    if (!PyArg_ParseTuple(args, "s", &myString))
        return NULL;
    myInt = myFunction(myString);
    return Py_BuildValue("i", myInt);
}
```

The function (myLib_myFunction in this case) definition always has two arguments (self and args) and there is a straightforward translation from the argument list in Python to the args argument.The args argument is a pointer to a Python tuple object

containing the arguments. Each item of the tuple corresponds to an argument in the call's argument list.

The arguments are Python objects, so in order to handle them we have to convert them to C values. The function PyArg_ParseTuple() in the Python API carries out this task. It checks the argument types and converts them to C values. PyArg_ParseTuple() returns true (nonzero) if all arguments have the right type and its components have been stored in the variables whose addresses are passed. It returns false (zero) if an invalid argument list was passed. In the latter case it also raises an appropriate exception so the calling function can return NULL immediately. If everything goes well, the string value of the argument gets copied to the local variable "myString".

The function Py_BuildValue() is something like the inverse of PyArg_ParseTuple(): it takes a format string and an arbitrary number of C values, and returns a new Python object. In this case, it will return an integer object.

If the C function returns no useful argument (void), the corresponding Python function must return None. To that extent, the following idiom is needed to do so.

```
Py_INCREF(Py_None);
return Py_None;
```

Py_None is the C name for the special Python object None. It is a genuine Python object rather than a NULL pointer, which means "error" in most contexts.

To make the C function callable from Python, it should be listed (name and address) in a "method table" as the following.

```
static PyMethodDef myLibMethods[] = {
    ...
    {"myFunction",  myLib_myFunction, METH_VARARGS, "Function
 description"},
    ...
    {NULL, NULL, 0, NULL}  /* Sentinel */
};
```

METH_VARARGS is a flag telling the interpreter the calling convention to be used for the C function. It should normally always be "METH_VARARGS" or "METH_VARARGS | METH_KEYWORDS". When using only METH_VARARGS, the function should expect the Python-level parameters to be passed in as a tuple acceptable for parsing via PyArg_ParseTuple(). The METH_KEYWORDS bit may be set in the

third field if keyword arguments should be passed to the function. In this case, the C function should accept a third PyObject * parameter which will be a dictionary of keywords. PyArg_ParseTupleAndKeywords() is used to parse the arguments to such a function.

Then, the above method table must be passed to the interpreter in the module's initialization function. The initialization function must be named initname(), where "name" is the name of the module, and should be the only non-static item defined in the module file.

```
PyMODINIT_FUNC initmyLib(void)
{
    (void) Py_InitModule("myLib", myLibMethods);
}
```

PyMODINIT_FUNC declares the following: the function as void return type, any special linkage declarations required by the platform and declares the function as extern "C" (for C++).

When the Python program imports "myLib" for the first time, initmyLib() is called. It calls Py_InitModule(), which creates a "module object" (which is inserted in the dictionary sys.modules under the key "myLib"), and inserts built-in function objects into the newly created module based upon the table (an array of PyMethodDef structures) that was passed as its second argument. Py_InitModule() returns a pointer to the module object that it creates. It may abort with a fatal error for certain errors, or return NULL if the module could not be initialized satisfactorily.

When embedding Python, the initmyLib() function is not called automatically unless there's an entry in the _PyImport_Inittab table. Therefore we should statically initialize it by directly calling initmyLib() after the call to Py_Initialize() as showed in the following.

```
int main(int argc, char *argv[])
{
    /* Pass argv[0] to the Python interpreter */
    Py_SetProgramName(argv[0]);

    /* Initialize the Python interpreter.  Required. */
    Py_Initialize();

    /* Add a static module */
```

```
    initmyLib();
    ...
}
```

Finally, to use the extension its necessary compiling and linking it with the Python interpreter. Two options are possible: dynamic loading and making the module a permanent part of the Python interpreter.

To make the module a permanent part of the Python interpreter it is necessary to change the configuration setup and rebuild the interpreter. On Unix systems, the file myLibModule.c shall be placed in the Modules/ directory of an unpacked source distribution, adding a line to the file Modules/Setup.local describing the file ("myLib myLibModule.o") and rebuild the interpreter.

For dynamic loading, which is the case of interest in PRISTINE, Python provides the "distutils" tool for building dynamically-linked extensions and custom interpreters. "distutils" is required to be installed on the build machine.

A distutils package contains a driver script, setup.py, which is a plain Python file that looks like the following.

```
from distutils.core import setup, Extension

module1 = Extension('myLibMocule', sources = ['myLibModule.c'])

setup (name = 'PackageName',
       version = '1.0',
       description = 'Package description',
       ext_modules = [module1])
```

Then, to compile myLibModule.c and produce an extension module named "myLibModule" in the build directory, execute the following:

```
$ python setup.py build
```

The module file will end up in a subdirectory build/lib.system, and will have a name like myLibModule.so or myLibModule.pyd (depending on the system).

In case additional preprocessor defines and libraries are needed, the file should look like the following.

**setup.py**

```
from distutils.core import setup, Extension

module1 = Extension('myLibModule',
                    define_macros = [('MAJOR_VERSION', '1'),
                                     ('MINOR_VERSION', '0')],
                    include_dirs = ['/usr/local/include'],
                    libraries = ['tcl83'],
                    library_dirs = ['/usr/local/lib'],
                    sources = ['myLibModule.c'])

setup (name = 'PackageName',
       version = '1.0',
       description = 'Package description',
       author = 'Name Surname',
       author_email = 'name@company.com',
       url = 'https://docs.python.org/extending/building',
       long_description = '''Long package description''',
       ext_modules = [module1])
```

In this example, setup() is called with additional meta-information: preprocessor defines, include directories, library directories, and libraries.

Once an extension has been successfully build, there are three ways to use it.

Installing the module,

```
python setup.py install
```

producing source packages,

```
python setup.py sdist
```

or including it in a source distribution through a MANIFEST.in file (not in the scope of PRISTINE).

## Language bindings with Python

So far we have seen how C/C++ functions can be called from Python. Now we will focus on the case in which we have an external library written in C/C++ that contains the functions that we want to call.

Assuming the function that we want to call from Python is located in the following C file

**myCFile.c**

```
#include <stdio.h>

void myFunction(myString)
{
    printf(myString);
    return;
}
```

The process of generating the needed glue code is the same as the one required to call C/C++ functions from Python code. The myLibModule.c would be part of the glue code, along all the other steps described along it.

The only change is that we should include the C file in the myLibModule.c file by means of the #include "myCFile.c" statement.

## 6.2. Automatic software wrapping using SWIG

SWIG (The Software Wrapper and Interface Generator) [swig] is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages and vice-versa. SWIG is used with different types of target languages including common scripting languages such as Perl, PHP, Python, Tcl and Ruby. The list of supported languages also includes non-scripting languages such as C#, Common Lisp (i.e. CLISP, Allegro CL, CFFI and UFFI), D, Go language, Java including Android, Lua, Modula-3, OCAML, Octave and R. Also several interpreted and compiled Scheme implementations (i.e. Guile, MzScheme/Racket and Chicken) are supported.

SWIG is most commonly used to create high-level interpreted or compiled programming environments, user interfaces, and as a tool for testing and prototyping C/C++ software. SWIG is typically used to parse C/C++ interfaces and generate the 'glue code' required for the above target languages to call into the C/C++ code and vice-versa. SWIG can also export its parse tree in the form of XML and Lisp s-expressions. SWIG is free software and the code that SWIG generates is compatible with both commercial and non-commercial projects.

The wrappers SWIG generates are layered: the C/C++ declarations are bound to a C/C++ Low Level Wrapper (LLW) which in turn is connected, using the Native Interface (NI) semantics of the target language, to a High Level Wrapper (HLW). Both the LLW and HLW depend on the target language since they have to interact using different NIs (such as the JNI [java-ni], the Python API [python-api] etc.). Depending on the complexity

of the library interface, SWIG has to be opportunely driven in order to produce good HL wrappers (and therefore target language modules or libraries suitable for the end-user). These corrections usually apply over an additional file (the SWIG interface file, also called the ".i" file), which can easily become an additional management burden. The following figure shows a simple example of a C software module wrapping.

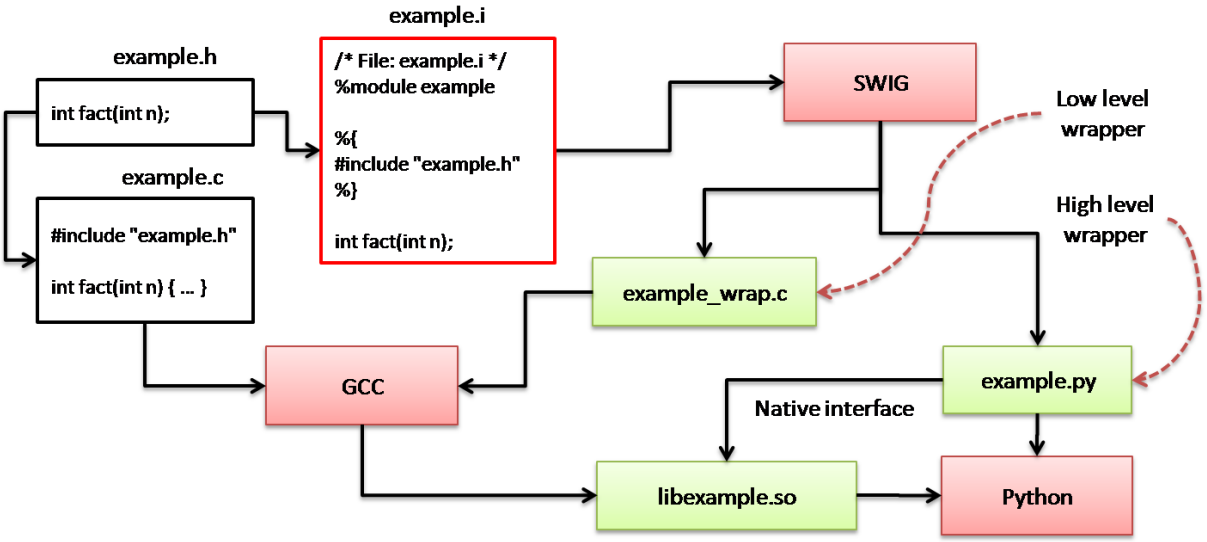


**Figure 12.**

In the example, the input software module (i.e. example.c and example.h) is exporting a function (i.e. the fact() function) and the relative SWIG driving directions for the binding productions are described in the interface file (i.e. the example.i file). Once the SWIG executable is executed with the interface file as input, it produces the low-level wrappers (i.e. example_wrap.c) and high-level wrappers (i.e example.py). Finally, The low-level wrappers are compiled as a dynamic library (i.e. libexample.so). This dynamic library will be loaded, on-demand, by the high-level wrappers once they are imported in the Python interpreter (i.e. Python).

For further information refer to the SWIG documentation [swig-documentation]. An overview of the SWIG general characteristics taken from the SWIG documentation is presented in the following.

### 6.2.1. The SWIG input

As input, SWIG expects a file containing ANSI C/C++ declarations and special SWIG directives. This input use to be the aforementioned interface file. The interface file contains ANSI C function prototypes and variable declarations. The %module directive defines the name of the module that will be created by SWIG. The %{ %} block provides a location for inserting additional code, such as C header files or additional C declarations, into the generated C wrapper code.

An example of the interface file would look like this.

**example.i**

```
%module example
%{
/* Put headers and other declarations here */
extern double My_variable;
extern int fact(int);
extern int my_mod(int n, int m);
%}
extern double My_variable;
extern int fact(int);
extern int my_mod(int n, int m);
```

### 6.2.2. The SWIG output

The output of SWIG is a C/C++ file that contains all of the wrapper code needed to build an extension module. SWIG may generate some additional files depending on the target language. By default, an input file with the name file.i is transformed into a file file_wrap.c or file_wrap.cxx.

The C/C output file created by SWIG often contains everything that is needed to construct an extension module for the target scripting language. To build the final extension module, the SWIG output file is compiled and linked with the rest of your C/C program to create a shared library.

### 6.2.3. The SWIG command

SWIG is invoked and run using the swig command, whose syntax is -swig [ options ] filename- (refer to <<swig-documentation for a full list of options). The swig command produces a new file called example_wrap.c that should be compiled along with the example.c file.

A general example to build C++ and Python bindings will be the following.

```
$ swig -c++ -python -outdir pyfiles -o cppfiles/example_wrap.cpp example.i
```

The name of the output file can be changed using the -o option. In certain cases, file suffixes are used by the compiler to determine the source language (C, C++, etc.). Therefore, you have to use the -o option to change the suffix of the SWIG-generated wrapper file if you want something different than the default. The -c++ option indicates

that the output file is a C++ file. The -outdir option modifies the default output directory (the same as the one for the generated C/C++ file).

The following commands would build a Python module from the above example interface file.

```
$ swig -python example.i
$ gcc -c -fpic example.c example_wrap.c -I/usr/local/include/python2.0
$ gcc -shared example.o example_wrap.o -o _example.so
```

Now we can call the fact() function from Python as follows:

```
$ python
>>> import example
>>> example.fact(4)
24
```

## 6.3. librina Java bindings

In the previous sections we have seen the basics of language bindings, both the manual approaches and the SWIG approach. In this section we will focus on the librina bindings for Java using SWIG.

The librina package contains the stack libraries that have been introduced to abstract the user from the kernel interactions (syscalls, Netlinks, etc.). Librina is more a framework/middleware than a library: it has its own memory model (explicit, no garbage collection), its execution model is event-driven and it uses concurrency mechanics (its own threads) to do part of its work.

C/C++ RINA programs in the user-space can use librina via statically/dynamically linkable libraries. I.e. as librina is a C++ library, C++ programs can make use of it by means of direct references.

Non-C/C++ RINA programs need language bindings to access the librina functionality. To that end, the glue code (or wrappers) to bind C/C++ with the target language need to be generated. As already explained, the approach followed by PRISTINE is to use SWIG to generate these wrappers. In the following, the approach to generate Java wrappers for librina using SWIG is explained.

It's important to note that, as also commented before, the librina wrappers are generated automatically upon system build. Therefore, the developer only needs to

keep the SWIG interface files up to date with respect to the respective C/C++ header files. The intention of this section is to explain the process of developing the said interface files so that developers can understand the interface files produced by PRISTINE, and if needed, can address future modifications and updates.

### 6.3.1. General aspects for Java bindings with SWIG

In this section we present some general aspects for the Java wrappers generation using SWIG. Some specifics must be addressed in the interface files for the correct wrapper generation. This section does not focus specifically on the librina bindings, but on relevant generic aspects needed for the correct interface file generation.

#### Type mappings

Since every programming language represents data differently, wrapper code for passing values between languages must be generated. In section 7.2 we described the default SWIG type mappings, which in general are enough for the general mapping cases. However, in case it's needed to modify the SWIG's default wrapping behavior for C/C++ datatypes, SWIG allows to do so by means of the %typemap directive. As a first example to illustrate this, consider the following interface file to map types between C and Python.

```
/* Convert from Python --> C */
%typemap(in) int {
    $1 = PyInt_AsLong($input);
}
/* Convert from C --> Python */
%typemap(out) int {
    $result = PyInt_FromLong($1);
}
```

"in" and "out" define the conversion direction. Special variables, prefixed with a $, are placeholders for C/C variables that are generated in the course of creating the wrapper function. $input refers to an input object that needs to be converted to C/C and $result refers to an object that is going to be returned by a wrapper function. $1 refers to a C/C++ variable that has the same type as specified in the typemap declaration (an int in this example). For further arguments in the typemap specification, consecutive numbering is used, i.e. $2, $3, etc.

An important aspect to consider is that once a typemap is defined, it is applied to all future occurrences of that type in the interface file.

In the following, this section presents the general approach to modify default typemaps for Java and the needed input in the SWIG interface files.

### Java Typemaps

A typemap is a code generation rule that is attached to a specific C datatype. The following example shows how to convert integer datatypes from Java to C, in which the function fact() is called by the Java code using a non-negative Java Integer value.

**example.i**

```
%module example
%typemap(in) int {
    $1 = $input;
    printf("Received an integer : %d\n", $1);
}
%inline %{
    extern int fact(int nonnegative);
%}
```

In this example, the "in" method refers to the conversion of Java input arguments to C/C++. The datatype int is the datatype to which the typemap will be applied. In the supplied code, the special variable (prefaced by a $) $1, is a placeholder for a local variable of type int. The $input variable contains the Java data, the JNI jint in this case.

When calling from Java the fact() function like the following:

```
System.out.println(example.fact(6));
```

The result showed would be:

```
Received an integer : 6
720
```

Which means that the supplied code in the %typemap directive is executed when converting the integer input value, and then, the result of the fact() function is returned.

In the above examples, the typemas track typenames (e.g. int, double, etc.). In addition, typemaps may also be specialized to match against a specific argument name or for sequences of consecutive arguments, like in the following example.

```
%typemap(in) double nonnegative {
```

```
    $1 = PyFloat_AsDouble($input);
    if ($1 < 0) {
       PyErr_SetString(PyExc_ValueError,"argument must be nonnegative.");
       return NULL;
    }
}
%typemap(in) (char *str, int len) {
    $1 = PyString_AsString($input); /* char *str */
    $2 = PyString_Size($input); /* int len */
}
...
double sqrt(double nonnegative);
int count(char *str, int len, char c);
```

## Exception handling

Part of the wrapping functionality is to map C function errors or C++ exceptions to Java exceptions. To this purpose, the %exception directive is used. The %exception directive allows to rewrite part of the generated wrapper code to include error check and throw Java Exceptions accordingly.

### C errors

In C functions, usually the returned errors are specified by returning a negative number or a null pointer. Let's consider the following example.

**example.i**

```
%exception malloc {
    $action
    if (!result) {
        jclass clazz = (*jenv)->FindClass(jenv, "java/lang/
OutOfMemoryError");
        (*jenv)->ThrowNew(jenv, clazz, "Not enough memory");
        return $null;
    }
}
void *malloc(size_t nbytes);
```

In this example, $action is a special variable which is replaced by the C/C++ function call being wrapped. The -return $null- line handles all native method return types (including void). When calling malloc() from Java with a large argument value, i.e. trying to allocate more memory that it's available, the exception produced by Java would look like:

```
Exception in thread "main" java.lang.OutOfMemoryError: Not enough memory
    at exampleJNI.malloc(Native Method)
    at example.malloc(example.java:16)
    at runme.main(runme.java:112)
```

If no declaration name is given to %exception, it is applied to all wrapper functions.

## C++ exceptions

To handle C++ exceptions, the process is to catch them and re-throw them as Java exceptions. Let's consider another example:

```
%javaexception("java.lang.Exception") getitem {
    try {
        $action
    } catch (std::out_of_range &e) {
        jclass clazz = jenv->FindClass("java/lang/Exception");
        jenv->ThrowNew(clazz, "Range error");
        return $null;
    }
}
class FooClass {
public:
    FooClass *getitem(int index); // Throws std::out_of_range exception
    ...
};
```

In this example, the %javaexception(classes) directive replaces %exception, where classes is a string containing one or more comma separated Java classes. Note that java.lang.Exception is a checked exception class and so ought to be declared in the throws clause of getitem().

## Renames

Renaming is used, for example, when the name of a C declaration generates a conflict with a keyword or already existing function in the target language, i.e. the names are the same, but the function in the target language is not related to the language bindings. The %rename directive is used to that extent. For example, the next example makes SWIG to call the C function print() when it's called by means of my_print() from the target language. In addition, it's also used to shorten a variable's long name.

```
%rename(my_print) print;
```

```
extern void print(const char *);
%rename(foo) a_really_long_and_annoying_name;
extern int a_really_long_and_annoying_name;
```

The %rename directive must be placed before the declarations to be renamed. %rename applies a renaming operation to all future occurrences of a name. The renaming applies to functions, variables, class and structure names, member functions, and member data. A common technique is to write code for wrapping a header file like this:

```
%rename(my_print) print;
%rename(foo) a_really_long_and_annoying_name;
%include "header.h"
```

Where the rename is applied to the functions included in header.h.

SWIG implements operator handling (mapping C/C++ operators into operators in the target language) in a general way using the %rename directive. For example, SWIG binds the Python's + operator to a method called *add* (which is the same name used to implement the Python + operator) in the following way.

```
%rename(__add__) Complex::operator+;
```

## 6.3.2. Librina interface file for Java

In section 7.2 we have explained the Automatic software wrapping using SWIG, presenting an example of a generic interface file. Here, we will address the specifics of the interface file needed for generating the Java wrappers. We will start by explaining the different parts of the file (for the complete version see [ref to librina.i]) and we will conclude with guidelines for the maintenance that these wrappings will need for future librina updates.

### Module, includes and conditional compilation statements

The first part of the interface file contains the module definition with the %module directive, the include statements with the %include directive, and the conditional compilation statements.

```
%module rina

%include "enums.swg"
%include <stdint.i>
```

```
%include <stl.i>

%include "stdlist.i"

#ifdef SWIGJAVA
#endif
```

The included file "enums.swg" is required to wrap all possible C/C++ enums using proper Java enums.

The SWIGJAVA symbol is defined when using Java, which is predefined by SWIG when it is parsing the interface. The #ifdef block allows to conditionally include parts of an interface (empty in the current interface file).

## Typemaps

Next, the typemaps are defined.

```
%typemap(jni)    void * "jbyteArray"
%typemap(jtype)  void * "byte[]"
%typemap(jstype) void * "byte[]"
%typemap(in)     void * {
    $1 = (void *) JCALL2(GetByteArrayElements, jenv, $input, 0);
}

%typemap(argout) void * {
    JCALL3(ReleaseByteArrayElements, jenv, $input, (jbyte *) $1, 0);
}

%typemap(javain) void * "$javainput"

%typemap(javaout) void * {
    return $jnicall;
}
```

%typemap(jni), %typemap(jtype) and %typemap(jstype) tell SWIG what JNI and Java types to use. %typemap(javain) and %typemap(javaout) handle the conversion of the jtype to jstype typemap type and vice-versa. %typemap(argout) is used to return values from arguments. It is combined with %typemap(in) to handle the $1 value.

## Exceptions

After this, the class Exception is defined.

```
%typemap(javabase) Exception "java.lang.Exception";
%typemap(javacode) Exception %{
  public String getMessage() {
    return what();
  }
%}
```

%typemap(javabase) and %typemap(javacode) are used for generating java code. The former sets the base for Java class and the latter copies Java code to the Java class.

And in the following, a set of typemaps for the rina exceptions are defined. All of them follow the same structure. For example, let's examine one of them.

```
%typemap(throws, throws="eu.irati.librina.IPCException")
 rina::IPCException {
  jclass excep = jenv->FindClass("eu/irati/librina/IPCException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
```

%typemap(throws, ...) is used to convert a C exception into an exception in Java. It provides a mechanism for handling the C methods that have declared the exceptions they will throw. To throw a checked exception, the 'throws' attribute is used specifying such exception.

The exceptions handled are the following:

- Exception
- IPCException
- FlowNotAllocatedException
- ReadSDUException
- WriteSDUException
- ApplicationRegistrationException
- ApplicationUnregistrationException
- FlowAllocationException
- FlowDeallocationException
- AllocateFlowException
- NotifyApplicationRegisteredException

- NotifyApplicationUnregisteredException
- NotifyFlowAllocatedException
- RegisterApplicationResponseException
- UnregisterApplicationResponseException
- AllocateFlowResponseException
- DeallocateFlowResponseException
- GetDIFPropertiesException
- GetDIFPropertiesResponseException
- AllocateFlowRequestArrivedException
- AppFlowArrivedException
- IpcmDeallocateFlowException
- NotifyFlowDeallocatedException
- InitializationException

### Typemaps to allow eventWait, eventPoll and eventTimedWait to downcast IPCEvent to the correct class

Let's have a look at the code that comes next.

```
%define DOWNCAST_IPC_EVENT_CONSUMER( OPERATION )
%typemap(jni) rina::IPCEvent *rina::IPCEventProducer::OPERATION "jobject"
%typemap(jtype) rina::IPCEvent *rina::IPCEventProducer::OPERATION
 "eu.irati.librina.IPCEvent"
%typemap(jstype) rina::IPCEvent *rina::IPCEventProducer::OPERATION
 "eu.irati.librina.IPCEvent"
%typemap(javaout) rina::IPCEvent *rina::IPCEventProducer::OPERATION {
    return $jnicall;
  }

%typemap(out) rina::IPCEvent *rina::IPCEventProducer::OPERATION {
    if ($1->eventType == rina::APPLICATION_REGISTRATION_REQUEST_EVENT) {
     rina::ApplicationRegistrationRequestEvent *appRegReqEvent =
 dynamic_cast<rina::ApplicationRegistrationRequestEvent *>($1);
        jclass clazz = jenv->FindClass("eu/irati/librina/
ApplicationRegistrationRequestEvent");
        if (clazz) {
            jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
            if (mid) {
                jlong cptr = 0;
```

```
                *(rina::ApplicationRegistrationRequestEvent **)&cptr =
 appRegReqEvent;
                $result = jenv->NewObject(clazz, mid, cptr, false);
            }
        }
    } else if ($1->eventType ==
 rina::APPLICATION_UNREGISTRATION_REQUEST_EVENT) {
     rina::ApplicationUnregistrationRequestEvent *appUnregReqEvent =
 dynamic_cast<rina::ApplicationUnregistrationRequestEvent *>($1);
        jclass clazz = jenv->FindClass("eu/irati/librina/
ApplicationUnregistrationRequestEvent");
        if (clazz) {
            jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
            if (mid) {
                jlong cptr = 0;
                *(rina::ApplicationUnregistrationRequestEvent **)&cptr =
 appUnregReqEvent;
                $result = jenv->NewObject(clazz, mid, cptr, false);
            }
        }
    }

    …

}
%enddef

DOWNCAST_IPC_EVENT_CONSUMER(eventWait);
DOWNCAST_IPC_EVENT_CONSUMER(eventPoll);
DOWNCAST_IPC_EVENT_CONSUMER(eventTimedWait);
```

The "…" represents the subsequents else if blocks accounting for different events with the same code structure than in the APPLICATION_UNREGISTRATION_REQUEST_EVENT as shown above.

The %define directive defines macros of code. Unlike normal C preprocessor macros, it is not necessary to terminate each line with a continuation character (\). Instead, the macro definition extends to the first occurrence of the %enddef directive.

When SWIG macros are expanded, they are reparsed through the C preprocessor. Thus, SWIG macros can contain all other preprocessor directives except for other nested macros (i.e. other %define directives).

When using arguments in the %define directive, variadic preprocessor macros are supported.

The following %typemap(jni), %typemap(jtype), %typemap(jstype) and %typemap(javaout) directives account for the same functionality already described in the Typemaps subsection above.

Next, the %typemap(out) directive maps C++ events to Java events. To do so, each if block checks if the event type corresponds to a certain event, performs a dynamic casting and tries to map the event to a Java event. If success, the event is returned with a reference to the (supposed to be defined) “init” method of the event.

The handled events are the following:

- APPLICATION_REGISTRATION_REQUEST_EVENT
- APPLICATION_UNREGISTRATION_REQUEST_EVENT
- FLOW_ALLOCATION_REQUESTED_EVENT
- FLOW_DEALLOCATION_REQUESTED_EVENT
- FLOW_DEALLOCATED_EVENT
- REGISTER_APPLICATION_RESPONSE_EVENT
- UNREGISTER_APPLICATION_RESPONSE_EVENT
- ALLOCATE_FLOW_RESPONSE_EVENT
- ALLOCATE_FLOW_REQUEST_RESULT_EVENT
- DEALLOCATE_FLOW_RESPONSE_EVENT
- GET_DIF_PROPERTIES_RESPONSE_EVENT

And finally, the %enddef directive indicates the end of the %define block, and the eventWait, eventPoll and eventTimedWait are allowed to downcast IPCEvent to the correct class.

## Renames

In the previous SWIG general aspects section we have introduced the %rename directive. Librina maps the RINA operators to the Java operators in the following way. Note that the includes are located after the %rename sentences so that the rename changes apply to the header contents.

```
%{
#include "librina/exceptions.h"
#include "librina/patterns.h"
#include "librina/concurrency.h"
#include "librina/common.h"
#include "librina/application.h"
```

```
%}

%rename(differs) rina::ApplicationProcessNamingInformation::operator!
=(const ApplicationProcessNamingInformation &other) const;
%rename(equals)
 rina::ApplicationProcessNamingInformation::operator==(const
 ApplicationProcessNamingInformation &other) const;
%rename(assign) rina::ApplicationProcessNamingInformation::operator=(const
 ApplicationProcessNamingInformation &other);
%rename(assign) rina::SerializedObject::operator=(const SerializedObject
 &other);
%rename(isLessThanOrEquals)
 rina::ApplicationProcessNamingInformation::operator<=(const
 ApplicationProcessNamingInformation &other) const;
%rename(isLessThan)
 rina::ApplicationProcessNamingInformation::operator<(const
 ApplicationProcessNamingInformation &other) const;
%rename(isMoreThanOrEquals)
 rina::ApplicationProcessNamingInformation::operator>=(const
 ApplicationProcessNamingInformation &other) const;
%rename(isMoreThan)
 rina::ApplicationProcessNamingInformation::operator>(const
 ApplicationProcessNamingInformation &other) const;
%rename(equals) rina::FlowSpecification::operator==(const
 FlowSpecification &other) const;
%rename(differs) rina::FlowSpecification::operator!=(const
 FlowSpecification &other) const;
%rename(equals) rina::Thread::operator==(const Thread &other) const;
%rename(differs) rina::Thread::operator!=(const Thread &other) const;
%rename(equals) rina::Parameter::operator==(const Parameter &other) const;
%rename(differs) rina::Parameter::operator!=(const Parameter &other)
 const;
%rename(equals) rina::Policy::operator==(const Policy &other) const;
%rename(differs) rina::Policy::operator!=(const Policy &other) const;
%rename(equals) rina::FlowInformation::operator==(const FlowInformation
 &other) const;
%rename(differs) rina::FlowInformation::operator!=(const FlowInformation
 &other) const;

%include "librina/exceptions.h"
%include "librina/patterns.h"
%include "librina/concurrency.h"
%include "librina/common.h"
%include "librina/application.h"
```

## Macro for defining collection iterators

The next part of the interface file is a macro to make Java iterators from C++ collections.

```
%define MAKE_COLLECTION_ITERABLE( ITERATORNAME, JTYPE, CPPCOLLECTION,
 CPPTYPE )
%typemap(javainterfaces) ITERATORNAME "java.util.Iterator<JTYPE>"
%typemap(javacode) ITERATORNAME %{
  public void remove() throws UnsupportedOperationException {
    throw new UnsupportedOperationException();
  }

  public JTYPE next() throws java.util.NoSuchElementException {
    if (!hasNext()) {
      throw new java.util.NoSuchElementException();
    }

    return nextImpl();
  }
%}
%javamethodmodifiers ITERATORNAME::nextImpl "private";
%inline %{
  struct ITERATORNAME {
    typedef CPPCOLLECTION<CPPTYPE> collection_t;
    ITERATORNAME(const collection_t& t) : it(t.begin()), collection(t) {}
    bool hasNext() const {
      return it != collection.end();
    }

    const CPPTYPE& nextImpl() {
      const CPPTYPE& type = *it++;
      return type;
    }
  private:
    collection_t::const_iterator it;
    const collection_t& collection;
  };
%}
%typemap(javainterfaces) CPPCOLLECTION<CPPTYPE> "Iterable<JTYPE>"
%newobject CPPCOLLECTION<CPPTYPE>::iterator() const;
%extend CPPCOLLECTION<CPPTYPE> {
  ITERATORNAME *iterator() const {
    return new ITERATORNAME(*$self);
  }
}
```

```
%enddef
```

The %typemap(javainterfaces) directive defines the Iterator interface. %typemap(javacode) overwrites the Iterator methods remove() and next() to throw a UnsupportedOperationException and return the next element respectively.

The %inline directive inserts code into the header of an interface file. The code is then parsed by both the SWIG preprocessor and parser. Note that it is illegal to include any SWIG directives inside a %{ ... %} block.

In the interface fie, the %inline directive is used to implement the hasNext() and nextImpl() methods, which check if the iterator is empty and returns the next element respectively.

Since SWIG cannot detect that the return value is a newly allocated object, the %newobject directive indicates to the target language that it should take ownership of the returned object.

Finally, the %extend directive is used to extend the C++ collection with the iterator constructor, which returns the iterator object.

## Iterator definition for C++ lists

The following part of the interface file makes use of the previous macro to define the iterators for different lists (which are C++ collections).

The code is the following.

```
MAKE_COLLECTION_ITERABLE(ApplicationProcessNamingInformationListIterator,
 ApplicationProcessNamingInformation, std::list,
 rina::ApplicationProcessNamingInformation);

MAKE_COLLECTION_ITERABLE(StringListIterator, String, std::list,
 std::string);

MAKE_COLLECTION_ITERABLE(FlowInformationListIterator, FlowInformation,
 std::list, rina::FlowInformation);

MAKE_COLLECTION_ITERABLE(UnsignedIntListIterator, Long, std::list,
 unsigned int);
```

As we can see in the second argument, the iterator definitions are made for the lists ApplicationProcessNamingInformation, String, FlowInformation and UnsignedInt.

## Templates

Template declarations are useless for SWIG, since it doesn't know how to generate any code until unless a definition of the class is provided.

The %template directive creates instantiations of a template class. %template() takes as argument the name of the instantiation in the target language. The name should not conflict with any other declarations in the interface file with one exception: the template name can match the name of a typedef declaration.

In the librina interface file, the template instantiations are the following.

```
%template(DIFPropertiesVector) std::vector<rina::DIFProperties>;
%template(FlowVector) std::vector<rina::Flow>;
%template(FlowPointerVector) std::vector<rina::Flow *>;
%template(ApplicationRegistrationVector)
 std::vector<rina::ApplicationRegistration *>;
%template(ParameterList) std::list<rina::Parameter>;
%template(ApplicationProcessNamingInformationList)
 std::list<rina::ApplicationProcessNamingInformation>;
%template(IPCManagerSingleton) Singleton<rina::IPCManager>;
%template(IPCEventProducerSingleton) Singleton<rina::IPCEventProducer>;
%template(StringList) std::list<std::string>;
%template(FlowInformationList) std::list<rina::FlowInformation>;
%template(UnsignedIntList) std::list<unsigned int>;
```

Whose template sentence syntax is the following.

%template (callable_name) class <data_type>

Callable_name accounts for the name referenced from the Java code, class accounts for the C++ template name and data_type accounts for the data type with which the template is instantiated.

# 7. Implementation status

This sections provides details regarding the features fully implemented and supported by the PRISTINE SDK, as released in the project first phase.

## 7.1. Supported policies

The following table reports the list of policy hooks already supported. Each row in the table refers to a different policy from the ones reported in PRISTINE D2.2. For a policy to be supported it means that it is possible for a plugin to define policy sets containing that policy, and that those policy-sets (with associated parameters) can be selected and configured using the *select-policy-set* and *set-policy-set-param* commands (in the IPC Manager console), accordingly to what reported in Section 5.1.1, “Overriding components' policies” and Section 5.2, “Policy-set selection”.

**Table 2. PRISTINE SDK supported policies**

| Policy name | Component | Research area | Notes |
|---|---|---|---|
| MaxQ Policy | RMT | Congestion control | Implemented. |
| RMTQ Monitor Policy | RMT | Congestion control | Implemented. |
| New Flow Request Policy | FA | Congestion control | Implemented. |
| New Member Access Control Policy | Security Manager | Auth., access control, confid. | Implemented. Current policy name is "*is Allowed To Join DIF*". |
| New Flow Access Control Policy | SecurityManager | Auth., access control, confid. | Implemented. Current policy name is "*acceptFlow*". |
| Rcvr Timer Inactivity Policy | DTP | Congestion control | Implemented. |
| Sender Inactivity Timer Policy | DTP | Congestion control | Implemented. |
| Initial Sequence Number Policy | DTP | Congestion control | Implemented. |

| Policy name | Component | Research area | Notes |
|---|---|---|---|
| Reconcile Flow Conflict Policy | DTCP | Congestion control | Implemented. |
| Receiving Flow Control Policy | DTCP | Congestion control | Implemented. |
| Rcvr Flow Control Policy | DTCP | Congestion control | Implemented. |
| NoRate-SlowDown Policy | DTCP | Congestion control | Implemented. |
| No Override Default Peak Policy | DTCP | Congestion control | Implemented. |
| Rate Reduction Policy | DTCP | Congestion control | Implemented. |
| Transmission Control Policy | DTP | Congestion control | Implemented, but ECN bit(s) not supported by IRATI. |
| Closed Window Policy | DTP | Congestion control | Implemented. |
| Flow Control Overrun Policy | DTP | Congestion control | Implemented. |
| Lost Control PDU Policy | DTCP | Congestion control | Implemented. |
| RTT Estimator Policy | DTCP | Congestion control | Implemented. |
| Retransmission Timer Expiry Policy | DTCP | Congestion control | Implemented. |
| Sender Ack Policy | DTCP | Congestion control | Implemented. |
| Receiving Ack List Policy | DTCP | Congestion control | Implemented. |
| Rcvr Ack Policy | DTCP | Congestion control | Implemented. |
| Sending Ack Policy | DTCP | Congestion control | Implemented. |
| Rcvr Control Ack Policy | DTCP | Congestion control | Implemented. |

| Policy name | Component | Research area | Notes |
|---|---|---|---|
| Rcvr Control Ack Policy | DTCP | Congestion control | Implemented. |
| DTCPpresent | DTP | Congestion control | Implemented. |
| Initial A-Timer | DTP | Congestion control | Implemented. |
| Sequence Number RollOver Threshold | DTP | Congestion control | Implemented. |
| Partial Delivery | DTP | Congestion control | Implemented. |
| Order | DTP | Congestion control | Implemented. Alternative name is "*In Order Delivery*". |
| Max allowable gap in SDUs | DTP | Congestion control | Implemented. |
| RMT Scheduling Policy | RMT | Resources Allocator | Implemented. Split into two policies, one for RX queues and one for TX queues, since the stack processes the queues in two different execution contexts. |
| PDU Forwarding Table generator policy | RA | Routing and addressing | Implemented. Computes PDU Forwarding Table based on the routing input. Current policy name is "*routing Table Updated*". |
| Address Assignment Policy | NSM | Routing and addressing | Implemented. Current policy name is "*get Valid Address*". |

| Policy name | Component | Research area | Notes |
|---|---|---|---|
| Address Validation Policy | NSM | Routing and addressing | Implemented. Current policy name is "*is Valid Address*". |
| Routing algorithm/ strategy | Routing | Routing and addressing | Implemented. All routing is policy, therefore the routing module is just a placeholder for the specific routing strategy. A policy based on link-state routing is currently available. |

## 7.2. Features planned for next release

The following subsections contain policies that are not yet supported by the first phase SDK. As outlined in the introduction of this document, missing policies will be supported in future SDK versions, and eventually released at M23 with PRISTINE D2.5. While not supported in the SDK, all these policies have been analyzed with respect to the IRATI implementation and PRISTINE D2.2 specifications, in order to

1. check if the associated components are supported by the IRATI stack
2. check if the policy specification (as reported by PRISTINE D2.2) is mature enough to proceed with an implementation
3. evaluate the feasibility of SDK support in terms of code changes

### 7.2.1. Resource Allocation research area

- *Flow Allocator, AllocateRetryPolicy*. FlowAllocatorInstance::createResponse() should call AllocateRetryPolicy, but the logic to resend another flow allocation request and restarting the flow creation timer is missing in the stack. Also it is not specified what are the information that a negative flow allocation response should carry to let the policy reformulate the request (e.g. what request parameter is not in range).

### 7.2.2. Authentication, access control, confidentiality research area

- *Security Manager, RIBAccessControlPolicy*. This policy could be invoked from RIBDaemon::writeObject(), RIBDaemon::readObject(), RIBDaemon::startObject(), RIBDaemon::stopObject(). Not specified what information is needed to check if a RIB operation can be accepted.
- *SDU protection, EncryptionPolicy*. The SDU protection component is not completely implemented by IRATI stack. It is can only be enabled/disabled system-wide at compile time, but more work is needed to implement it as a complete kernel component in order to make it programmable.
- *CACEP, AuthenticationPolicy*. Current IRATI does not distinguish this policy from the NewMemberAccessControlPolicy. Moreover, there is no support for authentication material (keys, passwords, certificates).

### 7.2.3. Resiliency and High Availability

- *Resource Allocator, MonitorNMinus1Flow*. Logic is missing in RA implementation - statistics, and statistics extraction. It will be easier to extract statistics in kernel-space. Moreover, this policy is only sketched, and not well defined. An alternative policy is FlowMonitoringPolicy.
- *Resource Allocator, NMinus1FlowDown*. Logic to detect that the flow is done is missing in the IRATI stack.
- *Flow Allocator, FlowMonitoringPolicy*. Should be implemented in kernel-space and preferred over MonitorNMinus1Flow. The definition and placement of this policy is still too immature to proceed with an implementation.

### 7.2.4. Congestion Control

- *Resource Allocator, CCA management*. Fine-grained policies regarding Congestion Control Avoidance have not been identified yet.
- *Resource Allocator, Resource Allocator behaviour*. This policy is still to be properly defined.

### 7.2.5. Routing and addressing

- *RMT, PDUForwardingPolicy*. Can be implemented accordingly to what reported in Section 5.1.1, “Overriding components' policies”.

### 7.2.6. Security Coordination

The Security Manager component should support the following policies:

- *CredentialManagementPolicy*
- *AuditingPolicy*
- *LoggingPolicy*

However, not enough details/specification - when the policies are invoked, what arguments are passed - are available to proceed with an implementation.

# 8. Conclusions and future works

The implementation of the first phase SDK described in this document required several extensions to the stack released by the FP7-IRATI project, but has been achieved without changing the IRATI High Level Architecture. A software framework has been created to allow plugins development and deployment. While being a Proof-of-concept SDK, several policies are already supported, that can already be used by WP3, WP4 and WP5 partners to validate the framework itself and provide feedback. Future SDK releases will address the policies that are yet not supported along with the implementation of the related components and functionalities that are still missing in the IRATI stack.

While the SDK framework is based on the C/C++ languages, this deliverable also addressed the design of bindings for high-level programming languages, in order to facilitate SDK usage, tests and experimentations. With reference to FP7-IRATI project, more language will be supported (Java, Python) and a broader portion of the librina APIs (librina-application, librina-cdap, librina-rib) will be accessible through the bindings.

Moreover, development activities to complement the IRATI stack with additional tools to support testing, debugging and integration activities have been started. Those tools are being addressed within the PRISTINE project and contributed as open-source software to the IRATI initiative:

- Configurator [open-irati-configurator]: The 'configurator' tool takes as input XML files that describe the network topology, the IPC processes on the different nodes in the network, the different DIFs in the network, and the applications that will be executed in the network. With this input, for each node, a configuration file is generated for the IPC Manager that will be running on the node. The 'configurator' tool will allow larger scale experiments to be performed, minimizing the time spent for scenario configuration and setup. This tool is being developed in the context of WP6.
- Valgrind [open-irati-valgrind]: Valgrind is a well-known tool for memory debugging, memory leak detection and profiling for user-space applications [valgrind]. In the IRATI stack the IPC Process and IPC Manager Daemons implementations are partially in user-space [irati-d31], as well as components developed within PRISTINE (e.g. Management Agent Daemon, user-space policies, APs) are in user-space, so valgrind is a very useful tool for debugging the

implementation. The valgrind software package contains the "vanilla" 3.10.0 valgrind's code-base patched with the support of IRATI's stack system calls.

# A. librina SWIG interface files

## stdlist.i

```
/*
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
 * MA  02110-1301  USA
 */

%{
#include <list>
%}

namespace std {

    template<class T> class list {
      public:
        typedef size_t size_type;
        typedef T value_type;
        typedef const value_type& const_reference;
        list();
        list(size_type n);
        size_type size() const;
        %rename(isEmpty) empty;
        bool empty() const;
        void clear();
        void reverse();
        %rename(addFirst) push_front;
        void push_front(const value_type& x);
        %rename(addLast) push_back;
        void push_back(const value_type& x);
        %rename(getFirst) front;
        const_reference front();
```

```
        %rename(getLast) back;
        const_reference back();
        %rename(clearLast) pop_back;
        void pop_back();
        %rename(clearFirst) pop_front;
        void pop_front();
        void remove(const value_type& x);
    };
}
```

## librina.i

```
/*
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
 * MA  02110-1301  USA
 */

%module rina

%include "enums.swg"
%include <stdint.i>
%include <stl.i>

%include "stdlist.i"

#ifdef SWIGJAVA
#endif

/**
 * void * typemaps.
 * These are input typemaps for mapping a Java byte[] array to a C void
 array.
 * Note that as a Java array is used and thus passeed by reference, the C
 * routine can return data to Java via the parameter.
```

```
 *
 * Example usage wrapping:
 *   void foo(void *array);
 *
 * Java usage:
 *   byte b[] = new byte[20];
 *   modulename.foo(b);
 */
%typemap(jni)    void * "jbyteArray"
%typemap(jtype)  void * "byte[]"
%typemap(jstype) void * "byte[]"
%typemap(in)     void * {
        $1 = (void *) JCALL2(GetByteArrayElements, jenv, $input, 0);
}

%typemap(argout) void * {
        JCALL3(ReleaseByteArrayElements, jenv, $input, (jbyte *) $1, 0);
}

%typemap(javain) void * "$javainput"

%typemap(javaout) void * {
        return $jnicall;
 }

/* Define the class Exception */
%typemap(javabase) Exception "java.lang.Exception";
%typemap(javacode) Exception %{
  public String getMessage() {
    return what();
  }
%}

%typemap(throws, throws="eu.irati.librina.Exception") Exception {
  jclass excep = jenv->FindClass("eu/irati/librina/Exception");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws, throws="eu.irati.librina.IPCException")
 rina::IPCException {
  jclass excep = jenv->FindClass("eu/irati/librina/IPCException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
```

```
%typemap(throws, throws="eu.irati.librina.FlowNotAllocatedException")
 rina::FlowNotAllocatedException {
  jclass excep = jenv->FindClass("eu/irati/librina/
FlowNotAllocatedException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws, throws="eu.irati.librina.ReadSDUException")
 rina::ReadSDUException {
  jclass excep = jenv->FindClass("eu/irati/librina/ReadSDUException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws, throws="eu.irati.librina.WriteSDUException")
 rina::WriteSDUException {
  jclass excep = jenv->FindClass("eu/irati/librina/WriteSDUException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws,
 throws="eu.irati.librina.ApplicationRegistrationException")
 rina::ApplicationRegistrationException {
  jclass excep = jenv->FindClass("eu/irati/librina/
ApplicationRegistrationException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws,
 throws="eu.irati.librina.ApplicationUnregistrationException")
 rina::ApplicationUnregistrationException {
  jclass excep = jenv->FindClass("eu/irati/librina/
ApplicationUnregistrationException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws, throws="eu.irati.librina.FlowAllocationException")
 rina::FlowAllocationException {
  jclass excep = jenv->FindClass("eu/irati/librina/
FlowAllocationException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
```

```
  return $null;
}
%typemap(throws, throws="eu.irati.librina.FlowDeallocationException")
 rina::FlowDeallocationException {
  jclass excep = jenv->FindClass("eu/irati/librina/
FlowDeallocationException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws, throws="eu.irati.librina.AllocateFlowException")
 rina::AllocateFlowException {
  jclass excep = jenv->FindClass("eu/irati/librina/
AllocateFlowException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws,
 throws="eu.irati.librina.NotifyApplicationRegisteredException")
 rina::NotifyApplicationRegisteredException {
  jclass excep = jenv->FindClass("eu/irati/librina/
NotifyApplicationRegisteredException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws,
 throws="eu.irati.librina.NotifyApplicationUnregisteredException")
 rina::NotifyApplicationUnregisteredException {
  jclass excep = jenv->FindClass("eu/irati/librina/
NotifyApplicationUnregisteredException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws, throws="eu.irati.librina.NotifyFlowAllocatedException")
 rina::NotifyFlowAllocatedException {
  jclass excep = jenv->FindClass("eu/irati/librina/
NotifyFlowAllocatedException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
```

```
%typemap(throws,
 throws="eu.irati.librina.RegisterApplicationResponseException")
 rina::RegisterApplicationResponseException {
  jclass excep = jenv->FindClass("eu/irati/librina/
RegisterApplicationResponseException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws,
 throws="eu.irati.librina.UnregisterApplicationResponseException")
 rina::UnregisterApplicationResponseException {
  jclass excep = jenv->FindClass("eu/irati/librina/
UnregisterApplicationResponseException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws, throws="eu.irati.librina.AllocateFlowResponseException")
 rina::AllocateFlowResponseException {
  jclass excep = jenv->FindClass("eu/irati/librina/
AllocateFlowResponseException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws,
 throws="eu.irati.librina.DeallocateFlowResponseException")
 rina::DeallocateFlowResponseException {
  jclass excep = jenv->FindClass("eu/irati/librina/
DeallocateFlowResponseException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws, throws="eu.irati.librina.GetDIFPropertiesException")
 rina::GetDIFPropertiesException {
  jclass excep = jenv->FindClass("eu/irati/librina/
GetDIFPropertiesException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws,
 throws="eu.irati.librina.GetDIFPropertiesResponseException")
 rina::GetDIFPropertiesResponseException {
```

```
  jclass excep = jenv->FindClass("eu/irati/librina/
GetDIFPropertiesResponseException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws,
 throws="eu.irati.librina.AllocateFlowRequestArrivedException")
 rina::AllocateFlowRequestArrivedException {
  jclass excep = jenv->FindClass("eu/irati/librina/
AllocateFlowRequestArrivedException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws, throws="eu.irati.librina.AppFlowArrivedException")
 rina::AppFlowArrivedException {
  jclass excep = jenv->FindClass("eu/irati/librina/
AppFlowArrivedException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws, throws="eu.irati.librina.IpcmDeallocateFlowException")
 rina::IpcmDeallocateFlowException {
  jclass excep = jenv->FindClass("eu/irati/librina/
IpcmDeallocateFlowException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws, throws="eu.irati.librina.NotifyFlowDeallocatedException")
 rina::NotifyFlowDeallocatedException {
  jclass excep = jenv->FindClass("eu/irati/librina/
NotifyFlowDeallocatedException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
}
%typemap(throws, throws="eu.irati.librina.InitializationException")
 rina::InitializationException {
  jclass excep = jenv->FindClass("eu/irati/librina/
InitializationException");
  if (excep)
    jenv->ThrowNew(excep, $1.what());
  return $null;
```

```
}


/* Typemaps to allow eventWait, eventPoll and eventTimedWait to downcast
 IPCEvent to the correct class */
%define DOWNCAST_IPC_EVENT_CONSUMER( OPERATION )
%typemap(jni) rina::IPCEvent *rina::IPCEventProducer::OPERATION "jobject"
%typemap(jtype) rina::IPCEvent *rina::IPCEventProducer::OPERATION
 "eu.irati.librina.IPCEvent"
%typemap(jstype) rina::IPCEvent *rina::IPCEventProducer::OPERATION
 "eu.irati.librina.IPCEvent"
%typemap(javaout) rina::IPCEvent *rina::IPCEventProducer::OPERATION {
    return $jnicall;
  }

%typemap(out) rina::IPCEvent *rina::IPCEventProducer::OPERATION {
    if ($1->eventType == rina::APPLICATION_REGISTRATION_REQUEST_EVENT) {
     rina::ApplicationRegistrationRequestEvent *appRegReqEvent =
 dynamic_cast<rina::ApplicationRegistrationRequestEvent *>($1);
        jclass clazz = jenv->FindClass("eu/irati/librina/
ApplicationRegistrationRequestEvent");
        if (clazz) {
            jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
            if (mid) {
                jlong cptr = 0;
                *(rina::ApplicationRegistrationRequestEvent **)&cptr =
 appRegReqEvent;
                $result = jenv->NewObject(clazz, mid, cptr, false);
            }
        }
    } else if ($1->eventType ==
 rina::APPLICATION_UNREGISTRATION_REQUEST_EVENT) {
     rina::ApplicationUnregistrationRequestEvent *appUnregReqEvent =
 dynamic_cast<rina::ApplicationUnregistrationRequestEvent *>($1);
        jclass clazz = jenv->FindClass("eu/irati/librina/
ApplicationUnregistrationRequestEvent");
        if (clazz) {
            jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
            if (mid) {
                jlong cptr = 0;
                *(rina::ApplicationUnregistrationRequestEvent **)&cptr =
 appUnregReqEvent;
                $result = jenv->NewObject(clazz, mid, cptr, false);
            }
        }
    } else if ($1->eventType == rina::FLOW_ALLOCATION_REQUESTED_EVENT) {
```

```
    rina::FlowRequestEvent *flowReqEvent =
 dynamic_cast<rina::FlowRequestEvent *>($1);
       jclass clazz = jenv->FindClass("eu/irati/librina/
FlowRequestEvent");
       if (clazz) {
           jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
           if (mid) {
               jlong cptr = 0;
               *(rina::FlowRequestEvent **)&cptr = flowReqEvent;
               $result = jenv->NewObject(clazz, mid, cptr, false);
           }
       }
   } else if ($1->eventType == rina::FLOW_DEALLOCATION_REQUESTED_EVENT) {
    rina::FlowDeallocateRequestEvent *flowReqEvent =
 dynamic_cast<rina::FlowDeallocateRequestEvent *>($1);
       jclass clazz = jenv->FindClass("eu/irati/librina/
FlowDeallocateRequestEvent");
       if (clazz) {
           jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
           if (mid) {
               jlong cptr = 0;
               *(rina::FlowDeallocateRequestEvent **)&cptr =
 flowReqEvent;
               $result = jenv->NewObject(clazz, mid, cptr, false);
           }
       }
   } else if ($1->eventType == rina::FLOW_DEALLOCATED_EVENT) {
    rina::FlowDeallocatedEvent *flowReqEvent =
 dynamic_cast<rina::FlowDeallocatedEvent *>($1);
       jclass clazz = jenv->FindClass("eu/irati/librina/
FlowDeallocatedEvent");
       if (clazz) {
           jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
           if (mid) {
               jlong cptr = 0;
               *(rina::FlowDeallocatedEvent **)&cptr = flowReqEvent;
               $result = jenv->NewObject(clazz, mid, cptr, false);
           }
       }
   } else if ($1->eventType == rina::REGISTER_APPLICATION_RESPONSE_EVENT)
 {
    rina::RegisterApplicationResponseEvent *flowReqEvent =
 dynamic_cast<rina::RegisterApplicationResponseEvent *>($1);
       jclass clazz = jenv->FindClass("eu/irati/librina/
RegisterApplicationResponseEvent");
       if (clazz) {
```

```
            jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
            if (mid) {
                jlong cptr = 0;
                *(rina::RegisterApplicationResponseEvent **)&cptr =
 flowReqEvent;
                $result = jenv->NewObject(clazz, mid, cptr, false);
            }
        }
    } else if ($1->eventType ==
 rina::UNREGISTER_APPLICATION_RESPONSE_EVENT) {
     rina::UnregisterApplicationResponseEvent *flowReqEvent =
 dynamic_cast<rina::UnregisterApplicationResponseEvent *>($1);
        jclass clazz = jenv->FindClass("eu/irati/librina/
UnregisterApplicationResponseEvent");
        if (clazz) {
            jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
            if (mid) {
                jlong cptr = 0;
                *(rina::UnregisterApplicationResponseEvent **)&cptr =
 flowReqEvent;
                $result = jenv->NewObject(clazz, mid, cptr, false);
            }
        }
    } else if ($1->eventType == rina::ALLOCATE_FLOW_RESPONSE_EVENT) {
     rina::AllocateFlowResponseEvent *flowReqEvent =
 dynamic_cast<rina::AllocateFlowResponseEvent *>($1);
        jclass clazz = jenv->FindClass("eu/irati/librina/
AllocateFlowResponseEvent");
        if (clazz) {
            jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
            if (mid) {
                jlong cptr = 0;
                *(rina::AllocateFlowResponseEvent **)&cptr = flowReqEvent;
                $result = jenv->NewObject(clazz, mid, cptr, false);
            }
        }
    } else if ($1->eventType == rina::ALLOCATE_FLOW_REQUEST_RESULT_EVENT)
 {
     rina::AllocateFlowRequestResultEvent *flowReqEvent =
 dynamic_cast<rina::AllocateFlowRequestResultEvent *>($1);
        jclass clazz = jenv->FindClass("eu/irati/librina/
AllocateFlowRequestResultEvent");
        if (clazz) {
            jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
            if (mid) {
                jlong cptr = 0;
```

```
                *(rina::AllocateFlowRequestResultEvent **)&cptr =
 flowReqEvent;
                $result = jenv->NewObject(clazz, mid, cptr, false);
            }
        }
    } else if ($1->eventType == rina::DEALLOCATE_FLOW_RESPONSE_EVENT) {
     rina::DeallocateFlowResponseEvent *flowReqEvent =
 dynamic_cast<rina::DeallocateFlowResponseEvent *>($1);
        jclass clazz = jenv->FindClass("eu/irati/librina/
DeallocateFlowResponseEvent");
        if (clazz) {
            jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
            if (mid) {
                jlong cptr = 0;
                *(rina::DeallocateFlowResponseEvent **)&cptr =
 flowReqEvent;
                $result = jenv->NewObject(clazz, mid, cptr, false);
            }
        }
    } else if ($1->eventType == rina::GET_DIF_PROPERTIES_RESPONSE_EVENT) {
     rina::GetDIFPropertiesResponseEvent *flowReqEvent =
 dynamic_cast<rina::GetDIFPropertiesResponseEvent *>($1);
        jclass clazz = jenv->FindClass("eu/irati/librina/
GetDIFPropertiesResponseEvent");
        if (clazz) {
            jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
            if (mid) {
                jlong cptr = 0;
                *(rina::GetDIFPropertiesResponseEvent **)&cptr =
 flowReqEvent;
                $result = jenv->NewObject(clazz, mid, cptr, false);
            }
        }
    }
}
%enddef

DOWNCAST_IPC_EVENT_CONSUMER(eventWait);
DOWNCAST_IPC_EVENT_CONSUMER(eventPoll);
DOWNCAST_IPC_EVENT_CONSUMER(eventTimedWait);

%{
#include "librina/exceptions.h"
#include "librina/patterns.h"
#include "librina/concurrency.h"
#include "librina/common.h"
```

```
#include "librina/application.h"
%}

%rename(differs) rina::ApplicationProcessNamingInformation::operator!
=(const ApplicationProcessNamingInformation &other) const;
%rename(equals)
 rina::ApplicationProcessNamingInformation::operator==(const
 ApplicationProcessNamingInformation &other) const;
%rename(assign) rina::ApplicationProcessNamingInformation::operator=(const
 ApplicationProcessNamingInformation &other);
%rename(assign) rina::SerializedObject::operator=(const SerializedObject
 &other);
%rename(isLessThanOrEquals)
 rina::ApplicationProcessNamingInformation::operator<=(const
 ApplicationProcessNamingInformation &other) const;
%rename(isLessThan)
 rina::ApplicationProcessNamingInformation::operator<(const
 ApplicationProcessNamingInformation &other) const;
%rename(isMoreThanOrEquals)
 rina::ApplicationProcessNamingInformation::operator>=(const
 ApplicationProcessNamingInformation &other) const;
%rename(isMoreThan)
 rina::ApplicationProcessNamingInformation::operator>(const
 ApplicationProcessNamingInformation &other) const;
%rename(equals) rina::FlowSpecification::operator==(const
 FlowSpecification &other) const;
%rename(differs) rina::FlowSpecification::operator!=(const
 FlowSpecification &other) const;
%rename(equals) rina::Thread::operator==(const Thread &other) const;
%rename(differs) rina::Thread::operator!=(const Thread &other) const;
%rename(equals) rina::Parameter::operator==(const Parameter &other) const;
%rename(differs) rina::Parameter::operator!=(const Parameter &other)
 const;
%rename(equals) rina::Policy::operator==(const Policy &other) const;
%rename(differs) rina::Policy::operator!=(const Policy &other) const;
%rename(equals) rina::FlowInformation::operator==(const FlowInformation
 &other) const;
%rename(differs) rina::FlowInformation::operator!=(const FlowInformation
 &other) const;

%include "librina/exceptions.h"
%include "librina/patterns.h"
%include "librina/concurrency.h"
%include "librina/common.h"
%include "librina/application.h"
```

```
/* Macro for defining collection iterators */
%define MAKE_COLLECTION_ITERABLE( ITERATORNAME, JTYPE, CPPCOLLECTION,
 CPPTYPE )
%typemap(javainterfaces) ITERATORNAME "java.util.Iterator<JTYPE>"
%typemap(javacode) ITERATORNAME %{
  public void remove() throws UnsupportedOperationException {
    throw new UnsupportedOperationException();
  }

  public JTYPE next() throws java.util.NoSuchElementException {
    if (!hasNext()) {
      throw new java.util.NoSuchElementException();
    }

    return nextImpl();
  }
%}
%javamethodmodifiers ITERATORNAME::nextImpl "private";
%inline %{
  struct ITERATORNAME {
    typedef CPPCOLLECTION<CPPTYPE> collection_t;
    ITERATORNAME(const collection_t& t) : it(t.begin()), collection(t) {}
    bool hasNext() const {
      return it != collection.end();
    }

    const CPPTYPE& nextImpl() {
      const CPPTYPE& type = *it++;
      return type;
    }
  private:
    collection_t::const_iterator it;
    const collection_t& collection;
  };
%}
%typemap(javainterfaces) CPPCOLLECTION<CPPTYPE> "Iterable<JTYPE>"
%newobject CPPCOLLECTION<CPPTYPE>::iterator() const;
%extend CPPCOLLECTION<CPPTYPE> {
  ITERATORNAME *iterator() const {
    return new ITERATORNAME(*$self);
  }
}
%enddef

/* Define iterator for ApplicationProcessNamingInformation list */
```

```
MAKE_COLLECTION_ITERABLE(ApplicationProcessNamingInformationListIterator,
 ApplicationProcessNamingInformation, std::list,
 rina::ApplicationProcessNamingInformation);
/* Define iterator for String list */
MAKE_COLLECTION_ITERABLE(StringListIterator, String, std::list,
 std::string);
/* Define iterator for Flow Information list */
MAKE_COLLECTION_ITERABLE(FlowInformationListIterator, FlowInformation,
 std::list, rina::FlowInformation);
/* Define iterator for Unsigned int list */
MAKE_COLLECTION_ITERABLE(UnsignedIntListIterator, Long, std::list,
 unsigned int);

%template(DIFPropertiesVector) std::vector<rina::DIFProperties>;
%template(FlowVector) std::vector<rina::Flow>;
%template(FlowPointerVector) std::vector<rina::Flow *>;
%template(ApplicationRegistrationVector)
 std::vector<rina::ApplicationRegistration *>;
%template(ParameterList) std::list<rina::Parameter>;
%template(ApplicationProcessNamingInformationList)
 std::list<rina::ApplicationProcessNamingInformation>;
%template(IPCManagerSingleton) Singleton<rina::IPCManager>;
%template(IPCEventProducerSingleton) Singleton<rina::IPCEventProducer>;
%template(StringList) std::list<std::string>;
%template(FlowInformationList) std::list<rina::FlowInformation>;
%template(UnsignedIntList) std::list<unsigned int>;
```

# References

[gpb] Google. 'Google Protocol Buffers developer guide'. Available online at https://developers.google.com/protocol-buffers

[irati-d21] IRATI project. 'D2.1 - First phase use cases updated RINA specifications and high-level software architecture'. Available online at http://irati.eu/wp-content/uploads/2012/07/IRATI-D2.1.pdf

[irati-d23] IRATI project. 'D2.3 Second phase use cases updated RINA specification and high-level software architecture'. Available http://irati.eu/wp-content/uploads/2012/07/IRATI-D2.3.zip

[irati-d31] IRATI project. 'D3.1 - First phase integrated RINA prototype over Ethernet for UNIX-like OS'. Available online at http://irati.eu/wp-content/uploads/2012/07/IRATI-D3.1-v1.0.pdf

[irati-d32] IRATI project. 'D3.2 - Second phase integrated RINA prototype over Ethernet for a UNIX-like OS'. Available online at http://irati.eu/wp-content/uploads/2012/07/IRATI-D3.2-v1.0.pdf

[irati-d33] IRATI project. 'D3.3 - Second phase integrated RINA prototype for Hypervisors for a UNIX-like OS'. Available online at http://irati.eu/wp-content/uploads/2012/07/IRATI-D3.3-bundle.zip

[irati-d34] IRATI project. 'D3.4 - Third phase integrated RINA prototype over Ethernet for a UNIX-like OS'. Available online at http://irati.eu/wp-content/uploads/2012/07/IRATI-D3.4.pdf

[json] Ecma International. 'The JSON Data Interchange Format'. Standard ECMA-404, October 2013. Available online at http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[python-api] The Python API – Available online at http://docs.python.org/2/extending/extending.html

[java-ni] The Java Native Interface – Available online at http://docs.oracle.com/javase/7/docs/technotes/guides/jni

[swig] SWIG - The Software Wrapper and Interface Generator - Available online at http://www.swig.org

[swig-documentation] SWIG - Documentation - Available online at http://www.swig.org/doc.html

[irati-ieee-network-magazine] Sander Vrijders; Dimitri Staessens; Didier Colle; Francesco Salvestrini; Eduard Grasa; Miquel Tarzan; Leonardo Bergesio. 'Prototyping the recursive internet architecture: the IRATI project approach'. IEEE Network Magazine, March 2014, Volume 28, Issue 2.

[module-init-tools] module-init-tools. Available online at http://ftp.kernel.org/pub/linux/utils/kernel/module-init-tools

[autoconf] Autoconf. Available online at http://www.gnu.org/software/autoconf

[automake] Automake. Available online at http://www.gnu.org/software/automake

[libtool] GNU Libtool - The GNU Portable Library Tool. Available online at http://www.gnu.org/software/libtool

[pkg-config] pkg-config. Available online at http://www.freedesktop.org/wiki/Software/pkg-config

[kbuild] Kbuild. Available online at https://www.kernel.org/doc/Documentation/kbuild/kbuild.txt

[kbuild-makefiles] Kbuild makefiles. https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt

[kconfig] Kconfig. Available online at https://www.kernel.org/doc/Documentation/kbuild/kconfig.txt

[pristine-d22] Pristine project. 'D2.2 - PRISTINE reference framework'

[valgrind] Valgrind. Available online at http://valgrind.org

[irati-home] The FP7 IRATI project. Available online at http://irati.eu

[protorina] ProtoRINA. Available online at https://github.com/ProtoRINA/users/wiki

[open-irati] The (Open) IRATI on GitHub. Available online at https://github.com/IRATI

[open-irati-valgrind] The (Open) IRATI's Valgrind. Available online at https://github.com/IRATI/valgrind

[open-irati-stack] The (Open) IRATI's Stack. Available online at https://github.com/IRATI/stack

[open-irati-traffic-generator] The (Open) IRATI's Traffic Generator. Available online at https://github.com/IRATI/traffic-generator

[open-irati-builder] The (Open) IRATI's Builder. Available online at https://github.com/IRATI/builder

[open-irati-wireshark] The (Open) IRATI's Wireshark. Available online at https://github.com/IRATI/wireshark

[open-irati-qemu] The (Open) IRATI's QEMU. Available online at https://github.com/IRATI/qemu

[open-irati-configurator] The (Open) IRATI's Configurator. Available online at https://github.com/IRATI/configurator

[libdl-online] The Open Group Base Specifications Issue 7. Available online at http://pubs.opengroup.org/onlinepubs/9699919799/functions/contents.html

[libdl-cpp-workaround] C++ dlopen mini HOWTO. Available online at http://www.tldp.org/HOWTO/C++-dlopen/

[boost-extension] Boost Extension library. Available online at http://boost-extension.redshoelace.com/docs/boost/extension/index.html

[kmod] kmod - Linux kernel module handling. Available online at http://git.kernel.org/cgit/utils/kernel/kmod/kmod.git/