Deliverable-2.5

Consolidated Software Development Kit

Deliverable Editor: Vincenzo Maffione, Nextworks s.r.l. (NXW)

Publication date:	30-November-2015
Deliverable Nature:	Software/Report
Dissemination level (Confidentiality):	PU (Public)
Project acronym:	PRISTINE
Project full title:	PRogrammability In RINA for European Supremacy of virTualIsed NEtworks
Website:	www.ict-pristine.eu
Keywords:	Software Development Kit, Policy sets, RINA Plugin Infrastructure, Programmability, Application Programming Interfaces, open IRATI RINA implementation
Synopsis:	This document describes the consolidated version 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.

List of Contributors

Deliverable Editor: Vincenzo Maffione, Nextworks s.r.l. (NXW)
NXW: Vincenzo Maffione, Gino Carrozzo
TSSG: Micheal Crotty
i2CAT: Eduard Grasa, Leonardo Bergesio
iMINDS: Sander Vrijders

Disclaimer

Executive Summary

This deliverable describes the consolidated version of the PRISTINE Software Development Kit (SDK). Task T2.3 has collected the feedbacks received from WP3, WP4, WP5 and WP6 during the PRISTINE first iteration, and has designed and implemented the SDK features required for the second iteration.

The RINA Plugin Interface (RPI) provides a set ot Application Programming Interfaces (APIs) that allow the SDK user to plug an unplug custom policies, and deploy them at run time. The RPI model defined during the first SDK release has proven to be effective in supporting the programmability needs of the technical work packages: a number of policies have been successfully integrated and deployed in the WP6 testbeds. As a consequence, no major model changes were required for the second iteration SDK.

The developments carried out by Task T2.3 have therefore focused on two main aspects. First, SDK support has been added for those components of the IPC Process not supported by the first SDK release. Second, a catalog of policies has been designed to automatically load plugins and keep track of the policies currently assigned to running IPC processes components.

This documents describes the updated version of the SDK, to be used for the project’s second iteration. In the next months, WP6 will integrate the policies developed by WP3, WP4 and WP5 - both new policies and consolidated version of existing policies - into the updated SDK framework.

Table of Contents

List of Figures

List of definitions

AP or DAP

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

CACEP

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

CDAP

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

CEP-id

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

DAF

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

DFT

Directory Forwarding Table. Sometimes referred to as search rules. Maintains a set of entries that map application naming information to IPC process addresses. The returned IPC process address is the address of where to look for the requested application. If the returned address is the address of this IPC Process, then the requested application is here; otherwise, the search continues. In other words, either this is the IPC process through which the application process is reachable, or may be the next IPC process in the chain to forward the request. The Directory Forwarding table should always return at least a default IPC process address

to continue looking for the application process, even if there are no entries for a particular application process naming information.

DIF

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

DTCP

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

DTP

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

DTSV

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

EFCP

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

FA

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

FAI

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

PCI

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

PDU

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

RA

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

RIB

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

RMT

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

SDU

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

1. 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

The FP7-IRATI project [irati-home] provided the first open-source kernel-space/user-space prototype [open-irati] of a RINA software stack. IRATI focused on the basic implementation of the different components of the IPC Process (IPCP) - e.g. Flow Allocator, Enrollment Task, Error and Flow Control Protocol (EFCP), etc - and basic related tools.

Taking the IRATI code-base as a starting point, PRISTINE has designed and implemented an open-source SDK framework to add support for programmability to the IRATI prototype, as required by the RINA architecture. Programmability allows SDK users to override the default behaviours (policies) of IPCP components - as they were implemented by IRATI - and provide alternative, customized behaviours that best fit the particular environment where the IPCPs are running. The SDK extension has been completely integrated into the IRATI code-base, without the need of changing its high level architecture. As a consequence, the IRATI deliverables illustrating architecture and implementation (e.g. [irati-d34]) are still a valid reference/manual for the IRATI stack. The IRATI implementation extended with the SDK is therefore publicly available on the GitHub repository [open-irati-stack].

Deliverable D2.3 [pristine-d23] reports the first version of the SDK, which allows for run-time plugging-in of policies in the form of policy-sets. Central to the SDK framework are the RPI model and concept of policy-set. Within the RPI model the different components of the IPCP and their policy sets are organized as a tree, where the child relationship models containment. A policy-set groups together different policies belonging to the same IPCP component, enabling easier cooperation (e.g. shared data) among those.

The first version of the SDK has been released, together with D2.3, through the pristine-1.1 branch of the GitHub repository. From that point, the SDK implementation has continuously evolved, adapting to the requirements of WP3, WP4 and WP5, and incorporating the feedbacks provided by WP6. The second iteration SDK, provided together with this document, is available through the pristine-1.3 branch of the GitHub repository. In the upcoming months, further integration and bug-fixing of the SDK code and other IRATI code will be provided by the pristine-1.4 branch, in terms of WP6 activities.

The ultimate purpose of the SDK is to allow developers to easily design, build and deploy custom policies for the RINA stack. These tasks must be possible without the need to deep into the internals of the IRATI implementation, and without the need to rebuild its code-base. In fact, the SDK provides developers with APIs to let them publish their policy-sets and let their policies to interact with the stack in the safest possible way. Moreover, policy-sets are packaged within plugins that are built externally to the IRATI code-base. Finally, SDK provides system administrator with tools to deploy plug-ins into a running stack.

Taking into account such requirements, the development efforts for the second phase SDK have striven for improving usability for developers and administrators, and for maximising coverage of supported IPCP components. A policy catalog has been implemented to allow automatic management of plugins and tracking of policy-sets. Build templates have been prepared for both user-space and kernel-space plugins to lower the learning barrier for new developers.

This deliverable is structured as follows. Section 2 describes the updates to the SDK high level architecture. Section 3 reports about the improved coverage of IPCP components and policies. Section 4 presents the policy management system of the consolidated SDK prototype. Section 5 contains tutorials for the development of user-space and kernel-space plugins. Section 6 reports the updated version of high-level programming language bindings, with reference to the ones reported in deliverable D2.3. Finally, section 7 provides conclusions and future works.

2. Updates to SDK architecture and implementation

The software architecture of the second iteration SDK comes with minor changes with respect to the first SDK version, released with D2.3. Major revisions were not necessary since the RPI model devised during the first PRISTINE iteration has proven to be adequate for the needs of the other WPs.

The following sections describe such minor architectural changes together with the updates to the internal SDK implementation.

2.1. Support for automatic default-filling of behavioural policies

As described in section 5.1 of [pristine-d23] (introduction of the RPI model), when a policy-set instance is created with some unspecified behavioural policies, the RPI has to provide suitable default ones.

This aspect of the RPI model, however, cannot always be implemented in a safe manner, because behavioural policies belonging to the same policy-sets can cooperate by means of a policy-set-specific shared state. In more detail, it is possible to distinguish two cases:

- If the default policy-set doesn’t use a policy-set-specific shared state, it means that all the interactions between its behavioural policies happen through the (IRATI) API exposed by the associated stack component.
- If the default policy-set uses a policy-set-specific shared state, it means that some of the behavioural policies in the policy-set access (read/write) the shared state to cooperate with other behavioural policies. Policy-set-specific shared state, by definition, is not exposed by the associated stack component.

The possible interactions among behavioural policies and the associated stack component are depicted in the following figure.

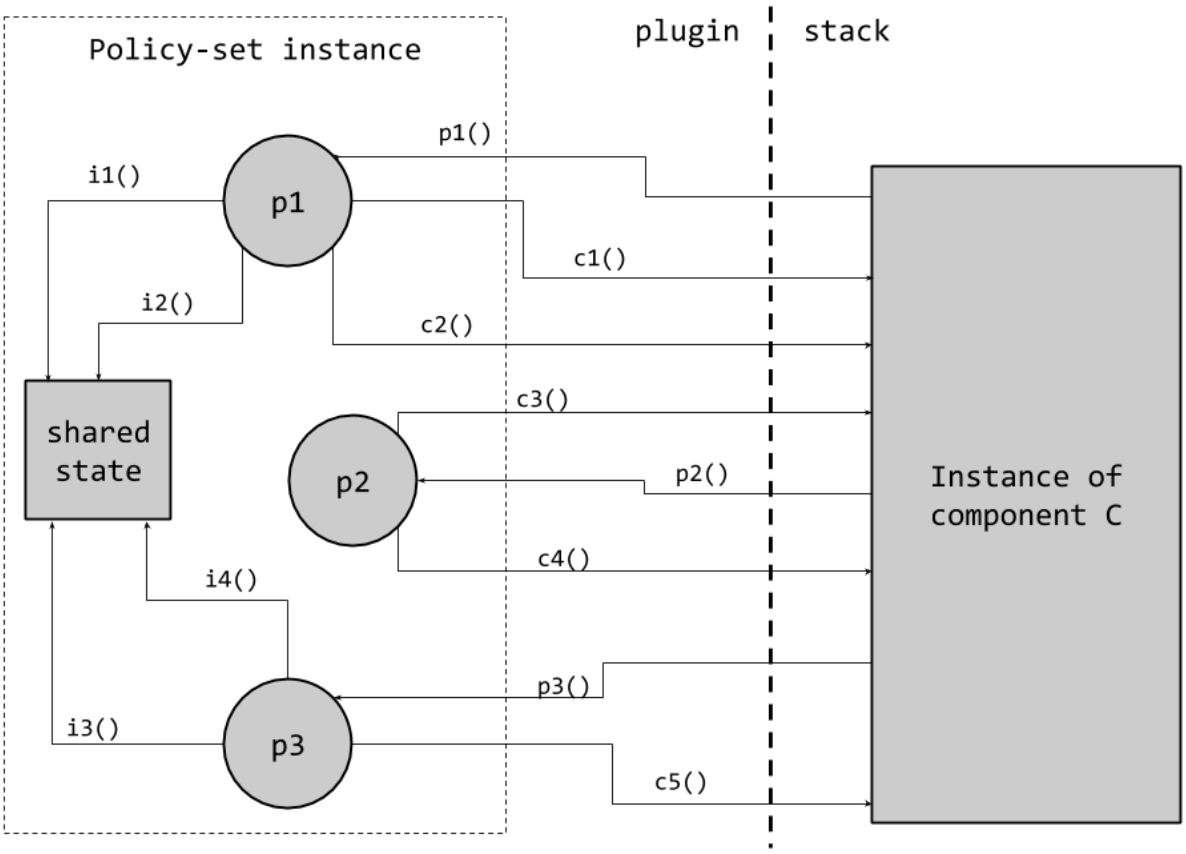

Figure 1. Example of interactions among a component instance and the policies of the associated policy-set instance.

`p1()`, `p2()` and `p3()` represent invocation of policies. `c1()`, `c2()`, …, represent invocation of the IRATI API exposed by the component. `i1()`, `i2()`, …, represent the policies accessing the policy-set-specific shared state. Note that while `p1` and `p3` access the shared state, `p2` doesn't.

If the default policy-set follows the first case, no private shared state is involved, and therefore it's safe for the RPI implementation to fill all the behavioural policies left unspecified with the corresponding default policies.

In the second case, the RPI implementation can - at least in theory - only default-fill those behavioural policies whose implementation doesn't access the private shared state. In practice, such default policies always access the private shared state, and therefore automatic default-filling is never possible.

Moreover, some behavioural policies can have a *null* implementation, meaning that no action has to be performed.

At the time of first iteration SDK this issue was not well understood, also because feedback from plugin developers was yet to come. Consequently the first iteration SDK didn't support automatic default-filling. Most of the default policies were simply exported by the SDK API, so

that the plugin developers could explicitly use them in the factory constructor for their policy-sets (manual default-filling).

The second iteration SDK, in contrast, provides support for automatic default-filling. The functions (or class methods) implementing default policies are statically built into the stack and not exported anymore through the SDK API. Each time a policy-set selection is triggered (see section 5.2 of [pristine-d23]) the RPI invokes a policy-set factory constructor to obtain a new policy-set instance. The new instance is checked for unspecified behavioural policies. When unspecified policies are found, decisions are taken depending on the type of associated component (e.g. DTP, RMT, Flow Allocator, etc.), using the following strategy:

- If the default policy-set for the involved component does not have a private shared state, unspecified policies are filled with the corresponding defaults. As an example, this is the case for the DTP and DTCP components.
- If the default policy-set for the involved component has a private shared state, there are two subcases:
  # if there is some unspecified behavioural policy that is mandatory, it's not possible to accommodate the policy-set selection request, which fails. As an example, this is the case for the RMT (dequeue/enqueue policies are mandatory) and PFF component (lookup/insert/remove policies are mandatory).
  # if all the unspecified behavioural policies are not mandatory, there is no need to deny the policy-set selection request, and no action will be performed when the component invokes the unspecified policy.

All the plugins published in the IRATI repository were updated to take advantage of the automatic default-fill support. In more detail, the policy-set factories constructors were updated to build policy-set instances with unspecified policies, so that the RPI implementation could perform proper sanity check, and possibly deny the policy-set selection request.

2.2. Support for policy-set selection transactions

As reported in section Section 2.1, policy-set selection operations can be denied because they request policy-sets with unspecified mandatory policies that cannot be default-filled. The component instance involved into the selection operation is in charge of performing this component-specific sanity check on the policy-set instance just created.

As a consequence, the policy-set selection workflow has slightly changed with reference to the first iteration SDK, as illustrated in the following figure.

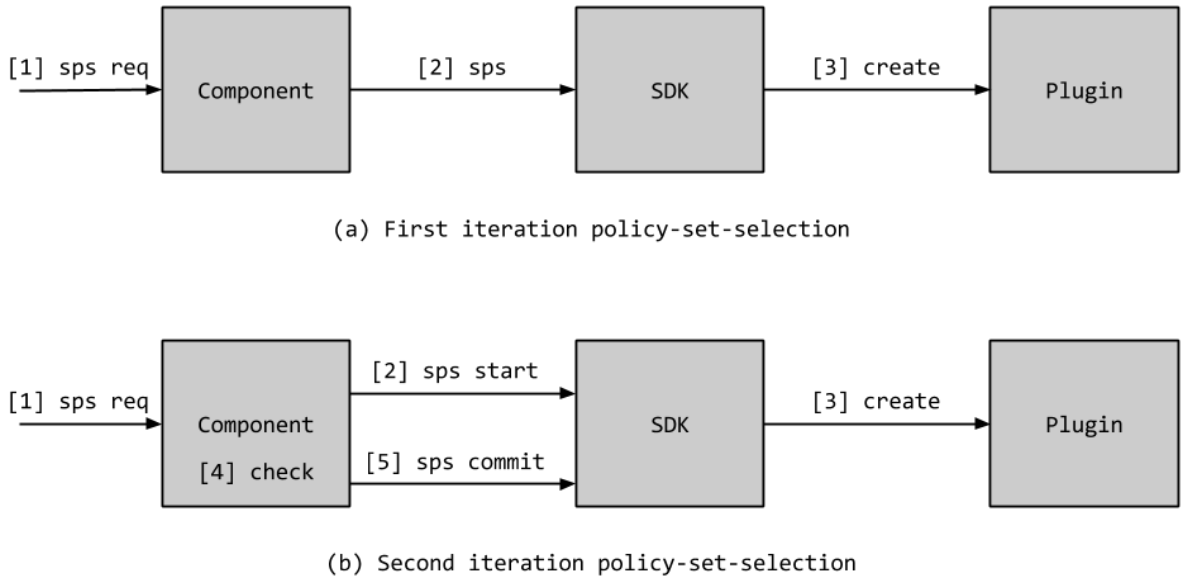

Figure 2. Policy-set selection workflows for the first and second iteration SDK

In the first iteration SDK (a), when a policy-set selection request (`sps`) is delivered to a component instance (1), this instance asks the SDK to create a new policy-set and install it (2). The SDK can therefore use the proper policy-set factory to create an instance (3) and then install it into the component instance.

In the second iteration SDK (b), the workflow is slightly more complex, to take into account the new requirements. When a policy-set selection request is delivered to a component instance (1), this instance asks the SDK to starts a select transaction (2). The SDK uses the proper policy-set factory to create an instance and returns it to the caller component (3). The caller can then perform the sanity check (4), that is different for different type of components, and announce its verdict to the SDK by committing the transaction (5). If the verdict is positive, the SDK will install the new policy-set instance. If the verdict is negative, the SDK will destroy the new policy-set instance and leave the old policy-set in place.

2.3. Updates to user-space RPI

2.3.1. Reorganisation of policy-set factories publishing API

In the first version of the uRPI interface, plugins were required to publish their policy-set factories using the `IPCProcess::psFactoryPublish()` interface, as reported in section 5.4.3 of [pristine-d23]. In detail, at plugin loading time the IPCP daemon was in charge of dynamically loading the plugin init() routine, that had to be implemented and exported by each plugin. The plugin was then required to call the IPCProcess::psFactoryPublish() method as many times as the number of policy-sets to publish.

This workflow was designed after the kRPI publishing API, mainly for the sake of uniformity. However, this choice resulted in a tricky internal implementation for the IPCP daemon software. In particular it was necessary to add an internal data structure to the IPCP class, to temporarily keep the (partial) list of policy-set factories loaded by the init() routine. Only after the termination of such routine, the temporary list could be flushed and the policy sets factories committed into the main policy-sets list (also internally kept by the IPCP class). This approach required executions of multiple init() routines to be serialized, since the temporary list was not thread-safe.

In addition to that, since the introduction of plugin manifests (see Section 4.3), the IPCP process knows in advance what policy-sets factories are supported by a certain plugin. This knowledge can be used by the IPCP daemon to load only a subset of the available plugins. Although it's not supported by the current implementation, it could even be possible to selectively load a proper subset of the policy-sets exported by a single plugin.

These aspects highlighted a better workflow for the policy-set publishing API, that consists in inverting the roles, as explained in the following. The init() routine has been replaced by the get_factories() routine, with the following signature:

```
extern "C" int get_factories(std::vector<struct rina::PsFactory>&
 factories);
```

At plugin loading time, the IPCP daemon dynamically loads and invokes the get_factories() routine, that all plugins must implement and export. The get_factories(), which is passed an empty list of policy-set factory objects (class rina::PsFactory), is in charge of filling the list with the factories to be exported. In other words, in contrast to the first iteration uRPI, in the second iteration the IPCP daemon takes the master role while the plugin initialisation routine takes the slave role. In this way, there is no need anymore to keep an internal temporary list of policy-set, and the thread-safety property comes for free as a consequence. The IPCProcess::psFactoryPublish() method, moreover, it's not needed anymore by plugins, and is therefore not exported. However, this method is still used by the internal IPCP implementation of the plugin loading procedures.

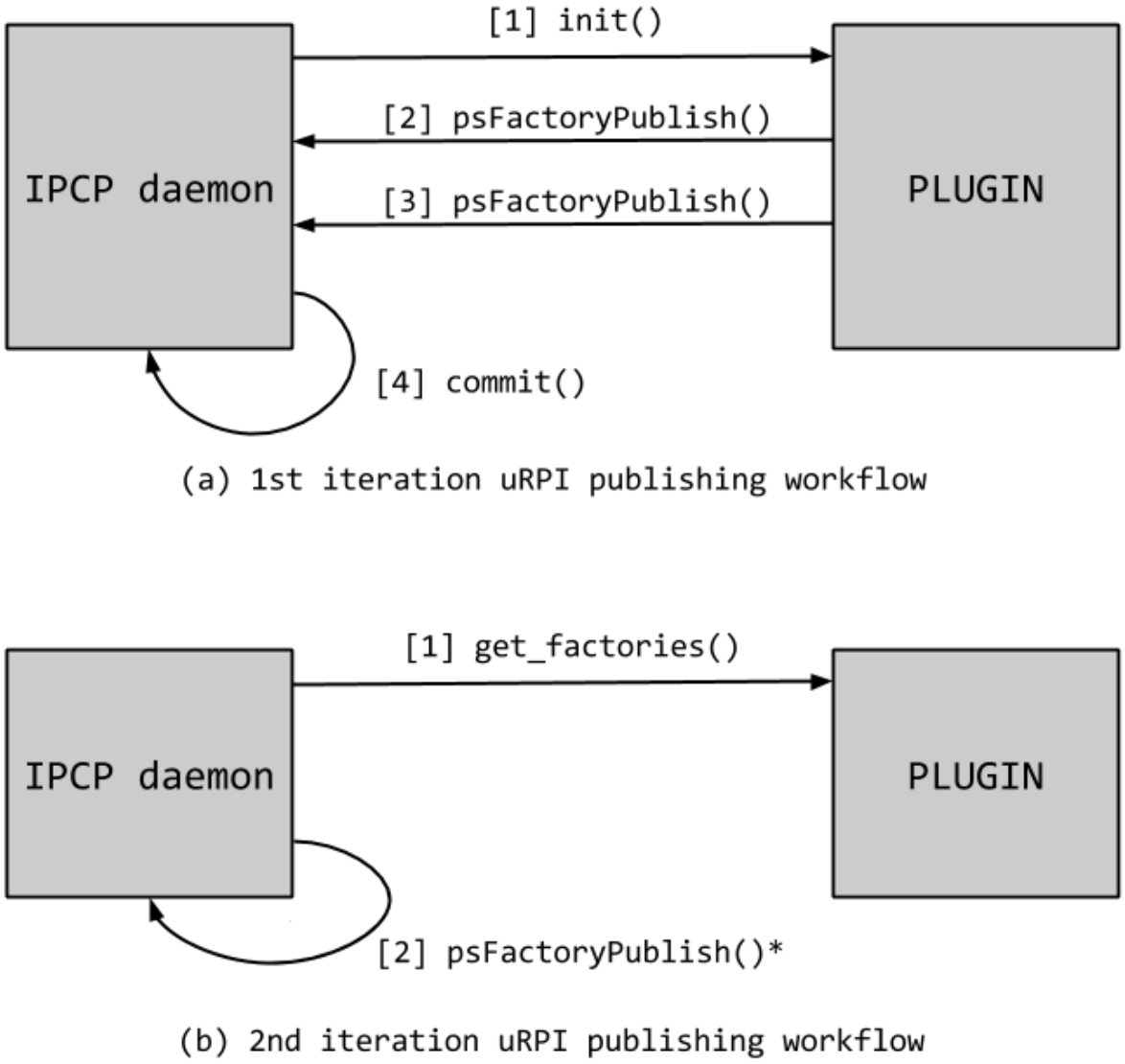

Figure 3. Workflows for publishing policy-set factories: comparison between first and second iteration SDK

The initialisation code of all the existing userspace plugins has been updated to comply with the new API.

2.3.2. Out-of-tree build of userspace plugins

Userspace plugins are built as dynamically loadable libraries (i.e. UNIX shared objects), to be loaded at run-time by the IPCP daemons. In the first iteration SDK, however, userspace plugins could only be built and packaged within the rinad build system. In other words, compiling and packaging an out-of-tree plugin (i.e. independently from rinad build system) was not supported by the SDK.

For an userspace plugin to be built, some rinad (internal) header files are necessary. Those headers contain the definitions and APIs of the userspace stack components, that are used by plugin developers to implement their policy-sets. When first iteration SDK was designed

and implemented, the status of those headers was judged not mature enough to export them. Moreover, since feedback from SDK users was yet to come, it was not clear what subset of the rinad internal APIs were to be exported. For these reasons, all the userspace plugins developed by PRISTINE were built and packaged within the rinad build system, where the required headers were already available.

As soon as the IPCP daemon codebase matured, and proper feedback was collected from SDK users, T2.3 extended the rinad build system to export the required headers, and made it possible to build out-of-tree userspace plugins.

Second iteration SDK provides the plugin developer with a template build system to support out-of-tree builds. In more detail, the `plugins/example-sm-plugin` directory in the github repository [open-irati] contains a (trivial) example plugin providing a policy-set for the Security Manager component of the IPCP. The example provides a Makefile that can be used to build and install the plugin. More information are reported in Section 5.1, which provides a tutorial to develop an userspace plugin.

2.4. Updates to kernel-space RPI

2.4.1. DTP/DTCP policy-set selection

SDK support for DTP and DTCP components has been available since the first iteration SDK. The following is an example IPCM console command to replace the policy-set for a DTCP component while the component is running (i.e. while packets are being transmitted and received)

```
IPCM >>> select-policy-set 3 efcp.18.dtcp red
```

As explained in Section 5.2 of [pristine-d23], this console command is a request for the IPCP with identifier 3 to select the "red" policy-set for an internal component of theirs, called efcp.18.dtcp. Since an IPCP supports many EFCP connections, there are many DTCP (and DTP) instances that can be addressed. In this example, the specific instance to be addressed is identified by "18".

In the first iteration SDK, this EFCP identifier was a connection-endpoint-id (cep-id), since a cep-id uniquely identifies an EFCP connection in the scope of an IPC Process, according to RINA specifications. A RINA flow, identified by a port-id, can be implemented by multiple EFCP connections, either in parallel or in sequence. While use of parallel EFCP connections has not been investigated yet, sequential connections can be used to match the bursty traffic patterns of certain applications, with each connection transporting a single data burst. Therefore,

as per RINA specifications, the lifespan of a flow is in general larger than the lifespan of one of its connections.

The current IRATI implementation is still not able to use multiple connection per flow (neither sequentially nor in parallel), and therefore the lifespan of flows and connections match.

From the point of view of policy-set selection, in any case, it is desirable to address flows (port-ids) more than connections (cep-ids). To understand the reason behind that, it should be noted that flows are terminated only after explicit flow deallocation, while connections could be terminated just because packets are not being exchanged on a flow for a while. Consequently, if a selection operation is performed on an EFCP connection, it will only affect to the current EFCP connection, and won’t have any effect on subsequent EFCP connections supporting the same flow. If the selection operation addresses the flow, instead, it means it addresses all the possible EFCP connections that will support the flow.

The second iteration SDK has therefore changed the meaning of DTCP/DTP component addressing, so that port-ids are used in place of cep-ids. In the example above, the meaning of the command would be to select the "red" policy set for the all the DTCP instances of the connections supporting the flow identified by port-id "18". Internally, the Component Delivery Workflow (section 5.4 of [pristine-d23]) will figure out the cep-id of the EFCP connection currently active on the specified flow.

3. Updates to SDK support

The PoC SDK delivered with D2.3 provided support for the majority of components and policies of the IPC Process. Some components/policies (see section 7.2 of [pristine-d23]), however, were left out because of time constraints or because the SDK-related requirements were not yet ready. The consolidated SDK delivered with this document completes the PoC SDK by adding coverage for (almost) all the IPCP components and policies.

The only component left out from the SDK is CDAP, where the programmability would come with the possibility to use different concrete or abstract syntaxes. However, task T2.3 came to the conclusion that such support is not useful within the objectives of PRISTINE. The Google Protocol Buffer concrete syntax (a choice inherited from IRATI) has proven to be a good choice, and no requirement in PRISTINE's scope would justify the effort to develop a different serialisation/deserialisation protocol.

The following sections report the updates occurred with respect to the first iteration SDK, in terms of SDK support of policies and components.

3.1. SDU protection

In the first iteration SDK, SDU protection was available only as a couple of compile-time global features. When compiling the IRATI kernel, the user could choose to include Cyclic Redundancy Check (CRC) and/or Time To Live (TTL) checks and processing. Once CRC/TTL were enabled, all the IPCPs used to apply those protections for all the connections.

The second iteration SDK comes with an improved implementation and support of the following SDU protection features:

- Error control, e.g. CRC.
- Encryption/decryption, e.g. DES, AES, etc.
- Packet lifetime checks, e.g. IP-like TTL.

These features are supported in the SDK, by means of three policy-sets.

```
struct sdup_errc_ps {
        struct ps_base base;

        /* Behavioural policies. */
```

```
        int (* sdup_add_error_check_policy)(struct sdup_errc_ps *,
                                            struct pdu_ser *);
        int (* sdup_check_error_check_policy)(struct sdup_errc_ps *,
                                              struct pdu_ser *);

        /* Reference used to access the SDUP data model. */
        struct sdup_port * dm;

        /* Data private to the policy-set implementation. */
        void *       priv;
};
```

The `sdup_add_error_check_policy()` is asked to compute an error-control code and store it in the serialised (i.e. wire-format) PDU. Conversely, the `sdup_check_error_check_policy()` is asked to extract the error-control code and check if it's correct, trying to detect if some bits were altered (and possibly what bits were altered). The default policy-set for the error control feature implements the CRC algorithm.

```
struct sdup_crypto_ps {
        struct ps_base base;

        /* Behavioural policies. */

        int (* sdup_apply_crypto)(struct sdup_crypto_ps *,
                                  struct pdu_ser *);
        int (* sdup_remove_crypto)(struct sdup_crypto_ps *,
                                   struct pdu_ser *);
        int (* sdup_update_crypto_state)(struct sdup_crypto_ps *,
                                         struct sdup_crypto_state *);

        /* Reference used to access the SDUP data model. */
        struct sdup_port * dm;

        /* Data private to the policy-set implementation. */
        void *       priv;
};
```

The `sdup_apply_crypto()` is in charge of performing cryptographic operations on the PDU - such as encrypting and/or adding a message authentication code (HMAC), while `sdup_remove_crypto()` tries to decrypt the PDU and/or check the message authentication code. Compression/decompression is also performed by these two operations if the policy considers it necessary. The `sdup_update_crypto_state()` is in charge of updating the key material used by the cryptographic routines (e.g. HMAC keys, session keys,

initialisation vectors, etc.). The default policy-set for the encryption/decryption SDU protection features uses the Linux kernel cryptographic libraries.

```
struct sdup_ttl_ps {
        struct ps_base base;

        /* Behavioural policies. */

        int (* sdup_set_lifetime_limit_policy)(struct sdup_ttl_ps *,
                                               struct pdu_ser *,
                                               size_t);
        int (* sdup_get_lifetime_limit_policy)(struct sdup_ttl_ps *,
                                               struct pdu_ser *,
                                               size_t *);
        int (* sdup_dec_check_lifetime_limit_policy)(struct sdup_ttl_ps *,
                                                     struct pdu *);

        /* Reference used to access the SDUP data model. */
        struct sdup_port * dm;

        /* Data private to the policy-set implementation. */
        void *       priv;
};
```

The `sdup_set_lifetime_limit_policy()` and `sdup_get_lifetime_limit_policy` are used to write and read a packet lifetime information to/from a serialised PDU. The `sdup_dec_check_lifetime_limit_policy()`, finally, is used to check if the lifetime of the serialised PDU has come to an end, and can be dropped. The default policy-set for the lifetime feature implements an IP-like TTL mechanism.

In the IPCP components tree, the error control, encryption and lifetime components are modeled as subcomponents of the SDU protection component, which is a subcomponent of the RMT in its turn. Consequently, the components are accessible - e.g. from IPCM console `set-policy-set` command - at component paths (see also 5.2.2 of [pristine-d23]) similar to the following ones:

- `rmt.sdup.63.errc`
- `rmt.sdup.63.crypto`
- `rmt.sdup.63.ttl`

where in the example 63 is the port-id of the N-1 port to be selected.

The SDU protection can be configured (see section Section 4.1) through the IPCM configuration file. An example of configuration is the following:

```
{
    "difType" : "normal-ipc",
    "securityManagerConfiguration" : {
     "policySet" : {
          "name" : "default",
          "version" : "1"
        },
        "authSDUProtProfiles" : {
           "default" : {
              "encryptPolicy" : {
                 "name" : "default",
                 "version" : "1",
                 "parameters" : [ {
              "name" : "encryptAlg",
              "value" : "AES128"
           }, {
              "name" : "macAlg",
              "value" : "SHA1"
           }, {
              "name" : "compressAlg",
              "value" : "default"
                } ]
              },
              "TTLPolicy" : {
                 "name" : "default",
                 "version" : "1",
                 "parameters" : [ {
                    "name" : "initialValue",
                    "value" : "50"
                  } ]
                },
                "ErrorCheckPolicy" : {
                   "name" : "CRC32",
                   "version" : "1"
                }
           }
        }
    }
  }
[...]
```

3.2. Authentication and Security Coordination

Authentication policies are used in the CACEP phase of flow allocation. The second iteration SDK supports CACEP with custom authentication policies between two N-IPCPs allocating an N-1 flow. The policies to be used can be selected by means of the IPCM configuration file, like in the following example.

```
{
    "difType" : "normal-ipc",
    "securityManagerConfiguration" : {
     "policySet" : {
          "name" : "default",
          "version" : "1"
        },
        "authSDUProtProfiles" : {
           "default" : {
              "authPolicy" : {
               "name" : "PSOC_authentication-ssh2",
           "version" : "1",
           "parameters" : [ {
              "name" : "keyExchangeAlg",
              "value" : "EDH"
           }, {
                    "name" : "keystore",
                    "value" : "/usr/local/irati/etc/private_key.pem"
                 }, {
                    "name" : "keystorePass",
                    "value" : "test"
                 } ]
              }
        }
    }
}
```

The following authentication policies are available in IRATI:

- No authentication (default)
- Password-based authentication
- Authentication based on the Secure-Shell (SSH) protocol

More information, together with an example of SDK usage, can be found in section Section 5.2.

3.3. Congestion Control

PRISTINE WP3 has used the SDK to implement congestion control algorithms by means of DTCP and RMT custom policy-sets. This is possible since DTCP and RMT were supported in the SDK since the first iteration. Congestion-related Resource Allocator policies are not included in the consolidated SDK since PRISTINE has not investigated them, and therefore no requirements, specifications or definitions were available.

3.4. PDU Forwarding Function

The first iteration PDU Forwarding Function (PFF) implementation was hardwired, only supporting link state routing with a an algorithm for next hop selection. However, alternative forwarding policies are possible - e.g. topological routing. The second iteration SDK has introduced full support for PFF policy-sets. In the IPCP component trees, the PFF has been modeled as a sub-component of the RMT, since PFF is used by the internal RMT implementation (all PFF operations are invoked by the RMT).

The policies included in the PFF policy-sets are reported and commented in the following C header excerpt:

```
struct pff_ps {
        struct ps_base base;

        /* Add a new PFF entry. */
        int (* pff_add)(struct pff_ps *        ps,
                        struct mod_pff_entry * entry);

        /* Remove an existing PFF entry. */
        int (* pff_remove)(struct pff_ps *        ps,
                           struct mod_pff_entry * entry);

        /* React to an N-1 port going down or going up. */
        int (* pff_port_state_change)(struct pff_ps * ps,
                                      port_id_t port_id, bool up);

        /* Check if the PFF is empty. */
        bool (* pff_is_empty)(struct pff_ps * ps);

        /* Flush all the PFF entries. */
        int  (* pff_flush)(struct pff_ps * ps);

        /* Return a next hop for a given destination address,
         * contained in the PCI header (@pci).
```

```
         * The lookup result is stored in the @ports/@count
         * output arguments. Multiple next-hops are supported. */
        int  (* pff_nhop)(struct pff_ps * ps,
                          struct pci *    pci,
                          port_id_t **    ports,
                          size_t *        count);

        /* Copy the contents of the PFF into a list, that
         * can be returned to user-space for inspection or
         * debugging. */
        int  (* pff_dump)(struct pff_ps *    ps,
                          struct list_head * entries);

        /* Reference used to access the PFF data model. */
        struct pff * dm;

        /* Data private to the policy-set implementation. */
        void *       priv;
};
```

In terms of resiliency and high-availability, the PFF implements the `pff_port_state_change` policy. This policy can be used to take actions in response to N-1 ports going down (or going up again). As an example, this is used in the WP4 Loop Free Alternates plugin to switch to alternatives next-hops ports as soon as the primary next-hop N-1 port goes down (fails).

3.5. Relaying and Multiplexing Task

The set of policies for the RMT component has been re-factored based on the feedbacks received from development of different RMT policy-sets (i.e. cherish/urgency, random early detection, jain binary scheme). These activities showed that the current set of policies for RMT was not convenient from an implementation point of view. This is due to the fact that, during the first iteration of the SDK, the set of policies was extracted directly from the RMT specifications, creating a one-to-one map between the policies described in the specifications and the hooks defined in the RMT policy set. The default implementation for these hooks were then filled with the functionalities that, before the SDK, were embedded in the RMT internal implementation. As a result, the hooks were poorly integrated with the RMT mechanisms and many times just not used. On the other side, the granularity of the functions to be performed by the RMT described in the specifications was not convenient for the implementation. As an example, although PDU monitoring, reacting to full queues and scheduling operations are conceptually separate steps of the same task (dispatching a PDU through a port), implementing them as three separate policies

was not the best choice, which required complex code to separate the implementation of those functionality.

In addition to reorganisation of RMT hooks, the RMT itself was modified to adapt to the new structure.

After experimentally identifying the previous issues, a new set of hooks was designed for the second iteration SDK. The next two code blocks show the old and new versions respectively. Finally, the specific changes are explained.

```
/* First iteration RMT policies. */

void (*max_q_policy_tx)(struct rmt_ps *,
   struct pdu *,
   struct rmt_n1_port *);
void (*max_q_policy_rx)(struct rmt_ps *,
   struct sdu *,
   struct rmt_n1_port *);
void (*rmt_q_monitor_policy_tx_enq)(struct rmt_ps *,
        struct pdu *,
        struct rmt_n1_port *);
void (*rmt_q_monitor_policy_tx_deq)(struct rmt_ps *,
        struct pdu *,
        struct rmt_n1_port *);
void (*rmt_q_monitor_policy_rx)(struct rmt_ps *,
    struct sdu *,
    struct rmt_n1_port *);
struct pdu *(*rmt_next_scheduled_policy_tx)(struct rmt_ps *,
         struct rmt_n1_port *);
int (*rmt_enqueue_scheduling_policy_tx)(struct rmt_ps *,
     struct rmt_n1_port *,
     struct pdu *);
int (*rmt_requeue_scheduling_policy_tx)(struct rmt_ps *,
     struct rmt_n1_port *,
     struct pdu *);
int (*rmt_scheduling_policy_rx)(struct rmt_ps *,
    struct rmt_n1_port *,
    struct sdu *);
int (*rmt_scheduling_create_policy_tx)(struct rmt_ps *,
           struct rmt_n1_port *);
int (*rmt_scheduling_destroy_policy_tx)(struct rmt_ps *,
     struct rmt_n1_port *);
```

```
/* Second iteration RMT policies. */
```

```
struct pdu *(*rmt_dequeue_policy)(struct rmt_ps *,
      struct rmt_n1_port *);
int (*rmt_enqueue_policy)(struct rmt_ps *,
            struct rmt_n1_port *,
     struct pdu *);
void *(*rmt_q_create_policy)(struct rmt_ps *,
      struct rmt_n1_port *);
int (*rmt_q_destroy_policy)(struct rmt_ps *,
              struct rmt_n1_port *);
```

1. `rmt_q_create_policy` and `rmt_q_destroy_policy` replace `rmt_scheduling_create_policy_tx` and `rmt_scheduling_destroy_policy_tx` but have the same functionality: they create a new internal RMT queue and private data within the RMT N-1 port. The `rmt_q_create_policy` returns a pointer to the new queue which will be treated as an opaque object by the RMT N-1 port, but used by the internal policy-set implementation. Storing a queue pointer in the N-1 port data structure allows quick access to the queue every time a PDU operation (enqueue/dequeue) is to be carried out.
2. `rmt_enqueue_policy` replaces the old `max_q_policy_tx`, `rmt_q_monitor_policy_tx_enq`, `rmt_enqueue_scheduling_policy_tx` and `rmt_requeue_scheduling_policy_tx`. It is therefore in charge of checking against queue overrun, monitoring the queue and inserting the PDU in a queue.
3. `rmt_next_scheduled_policy_tx` was replaced by `rmt_dequeue_policy` which reflects more clearly the function to be performed, being symmetric to the `rmt_enqueue_policy`.
4. The old receive-side policies, namely `max_q_policy_rx`, `rmt_next_scheduled_policy_rx` and `rmt_requeue_scheduling_policy_rx` have been discarded since the model used for the RMT's N-1 ports (part of the mechanism) does not consider queuing in reception. Received PDUs are directly pushed to EFCP.

Finally, for a better integration between RMT policies and the RMT core implementation, some return codes have been introduced for `rmt_enqueue_policy`, in order to notify the stack about the result of the operation and react accordingly:

- `RMT_PS_ENQ_SCHED`: PDU was enqueued properly, the stack must schedule (which currently means scheduling a kernel tasklet).

- `RMT_PS_ENQ_DROP`: N-1 port queue is full or for some other reason PDU has been dropped.
- `RMT_PS_ENQ_ERR`: Something is wrong (no queues founds for N-1 port, etc).

4. Policy management system

This chapter presents one of the main contribution of D2.5, that is the introduction of a *policy management system* within the consolidated SDK framework. The policy management system improves the SDK functionalities and usability for its users by automating those operations that used to be carried out manually with the first iteration SDK.

SDK users can be of categorised as follows:

- System/network administrators, who are human operators that use the SDK to install and load plugins, select policies and tune parameters of a DIF depending on the desired strategies and DIF operating environment, and monitor the status of plugins and policies.
- RINA management systems, which are distributed applications (possibly DAFs) aimed at performing automatically and in a coordinated manner the task of a system/network administrator.
- Plugin developers, that use the SDK API to implement policies in the form of policy-sets, and provide packaged plugins to the administrators and/or management systems.

4.1. Configuration file support

In the first SDK release, the only way to select non-default policies for IPCP components is to use the `select-policy-set` IPCM console command, as explained in section 5.2.3 of [pristine-d23]. When a RINA management system is in place, this kind of operations can also be automatically performed by the local Management Agent (MA), after receiving proper instructions from the RINA Manager. Both MA and IPCM console use the internal API exposed by the IPCM core to invoke the `select-policy-set` operation.

When an IPCP is configured with an `assign-to-dif` operation, therefore, default policy-sets are used for all the component instances of such IPCP. Subsequent `select-policy-set` operations can replace the defaults. However, it should be noted that an IPCP process becomes fully operative as soon as the `assign-to-dif` operation completes, and therefore default policy-sets will always be active for a finite amount of time (that can possibly be very small, if selection operations come from the IPCM immediately afterwards). Hence, an annoying race condition exist between the selection of custom-policy sets and other entities (N +1 applications, or neighbors N-IPCPs) interacting with the involved IPCP.

Moreover, if no automated management system controls the local RINA stack, the human administrator is obliged to carry out all the required `select-policy-set` operations manually, every time the system is rebooted.

The second SDK release addresses this concerns by adding configuration file support for the policy-set selection. The DIF configuration, stored in the IPCM configuration file, has been extended with "policySet" JSON records that can be used to specify the name of the policy-sets to be used and their versions, as illustrated in the following IPCM configuration file excerpt.

```
[...]
     "rmtConfiguration" : {
        "pftConfiguration" : {
          "policySet" : {
            "name" : "lfa",
            "version" : "1"
          }
        },
        "policySet" : {
          "name" : "red",
          "version" : "1"
        }
     },
     "flowAllocatorConfiguration" : {
         "policySet" : {
           "name" : "default",
           "version" : "2"
          }
     },
     "namespaceManagerConfiguration" : {
         "policySet" : {
           "name" : "default",
           "version" : "1"
           }
     },
[...]
```

In the example above, the "red" policy-set (implementing the Random Early Detection algorithm) is specified for the RMT component, the "lfa" policy-set (implementing the Loop Free Alternates algorithm) is specified for the PFF component, while default policy-sets are used for the Flow Allocator and Namespace Manager components.

When the IPCM sends the `assign-to-dif` operation to the IPCP process to be configured, the policy-set configuration records are sent alongside all the usual DIF configuration (e.g. DIF-wide constants, enrollment timeouts, etc.). In this way, the IPCP can initialize the policy-sets of its component contextually to the other IPCP configuration operations, in such a way that when the `assign-to-dif` operation completes the policy-sets specified by the configuration have already been activated. This removes the race condition present in the previous SDK release.

Since policy-set configuration is stored on a (persistent) configuration file, there is no need for the human administrator to use the IPCM console to specify the initial policy-sets. Selection operations via console may still be used in order to hot-replace policy-sets for running components.

4.2. DIF templates

At the time of the first SDK release, the IPCM configuration - which includes the configuration for all the DIFs and IPC Processes running on the local node - used to be contained in a single configuration file, (usually called `ipcmanager.conf`).

In the early PRISTINE experimentation trials, that involves configuration of many nodes, it was realised that a considerable amount of configuration was the same for all the nodes, with only some parts being different among them. Configuration sections describing installation paths (for binaries, libraries, plugins, etc.) and describing the IPCPs to be created (with name, type, DIF they belong to, etc.) are usually different across the nodes.

Sections describing DIF configurations (including data transfer constants, QoS cubes, policy configuration of IPCP components), instead, can be often shared between many nodes. Considering that these sections also collect all the policy-set-related configuration described in Section 4.1, the amount of configuration that can be shared has even increased with reference to first iteration SDK.

These observations constitute the motivation to introduce a template mechanism to share DIF configurations among different IPCM configuration files - and so among different nodes. A DIF template describes the configuration of a DIF - data transfer constants, QoS cubes, per-component policy-set configuration, etc. - and is stored in a separate file. The IPCM configuration file can refer to one or more DIF templates, in the sections describing the DIF configurations.

In the following example of IPCM configuration file

```
[...]
 "difConfigurations": [
        {
            "name": "300",
            "template": "shim-eth-vlan.dif"
        },
        {
            "name": "normal.DIF",
            "template": "default.dif"
        }
```

```
    ]
[...]
```

the DIF named "300" refers to a DIF template called "shim-eth-vlan.dif", while the DIF named "normal.DIF" refers to a DIF template called "default.dif".

An example of DIF template ("default.dif" in the previous excerpt) is the following one:

```
{
    "difType" : "normal-ipc",
    "dataTransferConstants" : {
     "addressLength" : 2,
     "cepIdLength" : 2,
     "lengthLength" : 2,
     "portIdLength" : 2,
     "qosIdLength" : 2,
     "sequenceNumberLength" : 4,
     "maxPduSize" : 10000,
     "maxPduLifetime" : 60000
    },
    "qosCubes" : [ {
  "name" : "unreliablewithflowcontrol",
        "id" : 1,
        "partialDelivery" : false,
        "orderedDelivery" : true,
        "efcpPolicies" : {
             "dtpPolicySet" : {
               "name" : "default",
               "version" : "0"
             },
             "initialATimer" : 300,
             "dtcpPresent" : true,
             "dtcpConfiguration" : {
                  "dtcpPolicySet" : {
                    "name" : "default",
                    "version" : "0"
                  },
                  "rtxControl" : false,
                  "flowControl" : true,
                  "flowControlConfig" : {
                      "rateBased" : false,
                      "windowBased" : true,
                      "windowBasedConfig" : {
                        "maxClosedWindowQueueLength" : 50,
                        "initialCredit" : 50
```

```
                    }
                }
            }
        }
    }, {
    "name" : "reliablewithflowcontrol",
      "id" : 2,
      "partialDelivery" : false,
      "orderedDelivery" : true,
      "maxAllowableGap": 0,
      "efcpPolicies" : {
           "dtpPolicySet" : {
             "name" : "default",
             "version" : "0"
           },
           "initialATimer" : 300,
           "dtcpPresent" : true,
           "dtcpConfiguration" : {
                "dtcpPolicySet" : {
                  "name" : "default",
                  "version" : "0"
                },
                "rtxControl" : true,
                "rtxControlConfig" : {
                    "dataRxmsNmax" : 5,
                    "initialRtxTime" : 1000
                },
                "flowControl" : true,
                "flowControlConfig" : {
                    "rateBased" : false,
                    "windowBased" : true,
                    "windowBasedConfig" : {
                      "maxClosedWindowQueueLength" : 50,
                      "initialCredit" : 50
                    }
             }
           }
      }
} ],
"knownIPCProcessAddresses" : [ {
 "apName" : "test1.IRATI",
 "apInstance" : "1",
 "address" : 16
}, {
 "apName" : "test2.IRATI",
 "apInstance" : "1",
```

```
    "address" : 17
  } ],
  "addressPrefixes" : [ {
    "addressPrefix" : 0,
    "organization" : "N.Bourbaki"
   }, {
    "addressPrefix" : 16,
    "organization" : "IRATI"
    } ],
   "rmtConfiguration" : {
      "pftConfiguration" : {
        "policySet" : {
          "name" : "default",
          "version" : "0"
        }
      },
      "policySet" : {
        "name" : "default",
        "version" : "1"
      }
   },
   "enrollmentTaskConfiguration" : {
      "policySet" : {
         "name" : "default",
         "version" : "1",
         "parameters" : [{
             "name"  : "enrollTimeoutInMs",
             "value" : "10000"
           },{
             "name"  : "watchdogPeriodInMs",
             "value" : "30000"
           },{
             "name"  : "declaredDeadIntervalInMs",
             "value" : "120000"
           },{
             "name"  : "neighborsEnrollerPeriodInMs",
             "value" : "30000"
           },{
             "name"  : "maxEnrollmentRetries",
             "value" : "3"
           }]
      }
   },
   "flowAllocatorConfiguration" : {
       "policySet" : {
         "name" : "default",
```

```
        "version" : "1"
      }
  },
  "namespaceManagerConfiguration" : {
      "policySet" : {
        "name" : "default",
        "version" : "1"
        }
  },
  "securityManagerConfiguration" : {
      "policySet" : {
        "name" : "default",
        "version" : "1"
        }
  },
  "resourceAllocatorConfiguration" : {
      "pduftgConfiguration" : {
        "policySet" : {
          "name" : "default",
          "version" : "0"
        }
      }
  },
  "routingConfiguration" : {
      "policySet" : {
        "name" : "link-state",
        "version" : "1",
        "parameters" : [{
          "name"  : "objectMaximumAge",
          "value" : "10000"
        },{
          "name"  : "waitUntilReadCDAP",
          "value" : "5001"
        },{
          "name"  : "waitUntilError",
          "value" : "5001"
        },{
          "name"  : "waitUntilPDUFTComputation",
          "value" : "103"
        },{
          "name"  : "waitUntilFSODBPropagation",
          "value" : "101"
        },{
          "name"  : "waitUntilAgeIncrement",
          "value" : "997"
        },{
```

```
            "name"  : "routingAlgorithm",
            "value" : "Dijkstra"
          }]
    }
  }
}
```

In conclusion, the decoupling of DIF templates from IPCM configuration resulted into more compact and less redundant IRATI stack configuration.

4.3. Plugin manifests

A prerequisite for the second iteration SDK is the existence of a mechanism for the IPC Manager to inspect the content of a plugin without explicitly (dynamically) loading it. This is useful for the policy catalog (see Section 4.4) and for the new userspace publishing API (Section 2.3).

In order to provide such a mechanism, plugins manifest were introduced. A manifest is an ASCII text file that accompanies the plugin code (a loadable kernel module for kernel-space plugins, or a shared object for user-space plugins). The JSON format, already used by the IPCM for its configuration file, was chosen to maximize code reuse.

A manifest file provides the following information

- The name of the associated plugin
- The version of the plugin
- The list of policy-sets exported by the plugin. For each policy-set, the manifest indicates
  # The name of the policy-set
  # The IPC Process component the policy-set applies to
  # The version of the policy-set

An example of manifest file for a plugin called `plugone` is the following:

```
{
        "PluginName": "plugone",
        "PluginVersion": "2",
        "PolicySets" : [
                {
                        "Name": "plsm1",
                        "Component": "security-manager",
                        "Version" : "1"
                },
```

```
                {
                        "Name": "plsm2",
                        "Component": "security-manager",
                        "Version" : "2"
                },
                {
                        "Name": "plnm",
                        "Component": "namespace-manager",
                        "Version" : "1"
                },
                {
                        "Name": "plra",
                        "Component": "resource-allocator",
                        "Version" : "2"
                },
                {
                        "Name": "plrt",
                        "Component": "routing",
                        "Version" : "1"
                }
        ]
}
```

There are naming constraints that must be met:

1. The name of the manifest file must have the `.manifest` extension (e.g. `plugone.manifest`)
2. The name of the manifest file and the name of the file containing the plugin object code must match, except for suffix (e.g. `plugone.manifest` and `plugone.so`)
3. The plugin name provided inside the manifest must match the name of the manifest file (and also the name of the object file), except for suffix

4.4. Policy catalog

The *policy catalog* is a central software component for the second iteration SDK, and it is implemented as a submodule of the IPC Manager Daemon. The policy catalog is in charge of the following tasks:

- Read the manifest files (see Section 4.3) for all the plugins installed on the local system.
- Automatically load all those plugins that are required to realise the policy configuration specified by the local IPCM configuration file. Loading a plugins means loading and publishing all the policy-sets exported by that plugin.

- Keep track of the plugins and policy-sets that are currently loaded.
- Keep track of the current mapping between running IPCP component instances and their associated policy-set.

4.4.1. Catalog database model

The catalog contains an internal database where all the relevant information is stored. The database contains three tables:

- The `Plugin` table contains an entry for each plugin installed in the system and known to the catalog. For each entry, the following information is maintained
  # The name of the plugin
  # The version of the plugin
  # The path where the plugin and its manifest is installed
  # The list of IPCPs for which this plugin has been loaded (initially empty)
- The `Policy-set` table contains an entry for each policy-set exported by some plugin represented in the Plugin table. For each entry, the following information is stored:
  # The name of the policy-set
  # The version of the policy-set
  # The IPCP component that the policy-set applies to (e.g. RMT, Flow Allocator, …)
  # A reference to the `Plugin` entry of the plugin that exported this policy-set
  # The list of resources for which this policy-set is used (initially empty)
- The `Resource` table contains an entry for each IPCP or RINA flow that has associated policy-sets. An IPCP is identified by an IPCP id, while a flow is identified by a port-id. For each entry, the following information is stored:
  # A reference to the `Policy-set` entry of the policy-set currently associated to this resource
  # The identifier for this resource, that is an IPCP id if the resource is an IPCP or a port-id if the resource is a flow

Note that the need to model both IPCP and flows as "resources" comes from the fact that some IPCP components are per-flow (DTP, DTCP, SDU Protection), while the other ones are per-IPCP (RMT, Flow Allocator, Security Manager, etc.).

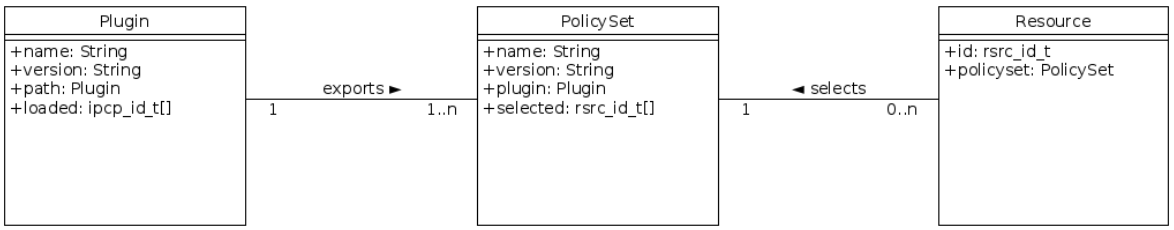

Figure 4. UML class diagram for the policy catalog database

4.4.2. Startup

At IPCM startup, the catalog performs an `import` operation, that consist in scanning some directories in the local filesystem to look for plugin manifest files. The directories to be scanned are specified in `localConfiguration` section of the IPCM configuration file, as shown in the following excerpt

```
[...]
"localConfiguration": {
        "installationPath": "/usr/local/irati/bin",
        "libraryPath": "/usr/local/irati/lib",
        "logPath": "/usr/local/irati/var/log",
        "consolePort": 32766,
        "pluginsPaths": [
                "/usr/local/irati/lib/rinad/ipcp",
                "/lib/modules/4.1.10-irati/extra"
        ]
    },
[...]
```

where the "pluginsPaths" variable specifies a list of paths to directories to be scanned. There is usually one path for the userspace plugins (a subdirectory in the rinad installation path), and a path for the kernelspace plugins (the kernel module installation path for the IRATI kernel).

Each path in "pluginsPaths" is scanned to look for files with `.manifest` extension. The files found are read to check syntax validity and their content is loaded in the database, in such a way to fill the `Plugin` table.

4.4.3. Assignment of an IPCP to a DIF

In order for an IPCP to become operative after creation, an `assign-to-dif` operation must be performed. The purpose of this operation is to provide the IPCP with all the configuration of the DIF it is part of. In more detail, the `assign-to-dif` operation takes a DIF template

(see Section 4.2) as an input, which includes the policy-set configuration for all the IPCP components.

The `load_by_template()` catalog method is invoked by the IPCM core while performing an `assign-to-dif` operation on an IPCP, with the DIF template being passed as an argument. `load_by_template()` takes care of loading all plugins that contain policy-sets referenced by the DIF template. As described in [pristine-d23], user-space plugins are dynamically loaded by the involved IPCP Daemon, while kernel-space plugins are loaded through a kernel module. As a consequence of the plugin loading, the `Policy-set` table gets populated, and the `Plugin` entries are updated to add the IPCP to the list, in order to track the fact that the plugins have been loaded for the involved IPCP.

Once the `load_by_template()` method completes, an `assign-to-dif` netlink message is sent the the IPCP daemon. Among the other things, the IPCP daemon carries out the policy-set selection for all its components. The names and versions of the policy-sets to be selected are contained in the netlink message, and because of the work executed by `load_by_template()`, at this point all those policy-sets have already been loaded.

As explained in the next subsection, policy-set selection operations have also the effect of updating the `Resource` table, adding entries if necessary.

An example of workflow for an `assign-to-dif` operation is illustrated in the following picture.

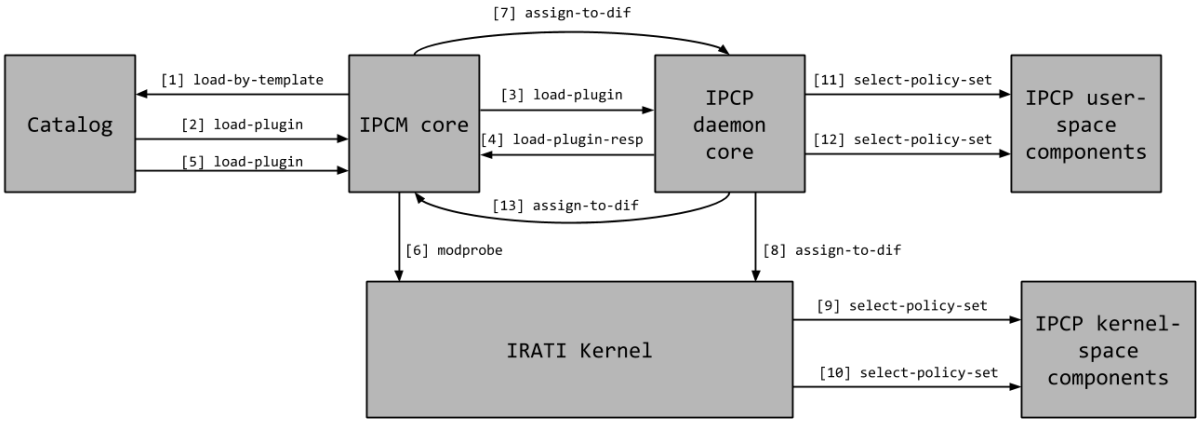

Figure 5. Workflow for an assign-to-dif

IPCM core starts the load-by-template procedure (1), which in turn loads an user-space plugin (2), (3) and (4), and a kernel-space plugin (5) and (6). Then the IPCM core sends an assign-to-dif message to the IPCP Daemon (7), which is propagated to kernel-space (8), where kernel-space policy-sets are selected (9) and (10). The IPCP Daemon completes the operation by selecting policy-sets for userspace components (11) and (12) and sends its response back to the IPCM core (13).

4.4.4. Policy-set selection

Once the `assign-to-dif` operation is completed, the IPCP becomes operational - it can issue and receive enrollment requests, flow allocation requests, local and remote RIB accesses, etc - with the initial policy-sets already in place.

However, a human administrator or a distributed management system may want to hot-replace policies to adapt to the evolving environment and requirements.

The IPCM core internal API exposes three policiy-related methods, as outlined in section 5.2 of [pristine-d23]:

- `plugin_load(string plugin_name, bool load)`, to load or unload an user-space or kernel-space plugin for the specified IPCP.
- `select_policy_set(int ipcp_id, string component_id, string ps_name)`, to hot-replace a policy-set for a component instance of an IPCP, as specified by the arguments.
- `set_policy_set_param(int ipcp_id, string component_id, string param_name, string param_value)`, to modify the value of a parameter of a component instance of an IPCP, as specified by the arguments.

These API calls are then exported through the IPCM console (for human administrators) and through the Management Agent (for distributed management systems).

In any case, when the IPCM core API receives a `select_policy_set()` call, the `load_policy_set()` catalog method is invoked. The catalog performs a lookup in the `Policy-set` table for the required policy-set. If there is no matching entry, it means that the policy-set is unknown (it was not exported by a manifest), and therefore the operation fails. If an entry is found, the associated `Plugin` entry is examined to check whether the plugin has already been loaded for the IPCP specified as an argument. If not, a `plugin_load` IPCM operation is performed.

Once the `load_policy_set()` completes, the `select_policy_set()` implementation sends a 'select-policy-set' netlink message to the involved IPCP Daemon, in order to trigger the Component Delivery Workflow (described in section 5.2.5 of [pristine-d23]) that carries out the selection. If the IPCM receives a corresponding netlink response message reporting success, the catalog is invoked again (`policy_set_selected()` method) in order to update the `Policy-set` and `Resource` tables. In more detail, an entry in the `Resource` table is created (if not already existing) for the resource specified by the `component_id`, and the involved `Policy-set` entry is updated to track that such resource uses the policy-set.

4.4.5. IPCM console commands

Two new catalog-related commands have been added to the IPCM console.

The `show-catalog` command dumps the current contents of the catalog database, listing all the `Policy-Set` entries with the indications about the related `Plugin` and `Resource` entries. The following listing shows an example of output

```
IPCM >>> show-catalog
Catalog of plugins and policy sets:
    ===================================
    ps name: default
    ps component: dtcp
    ps version: 1
    plugin: /home/vmaffione/irati/local/lib/rinad/ipcp/rina-default-plugin
    loaded for ipcps [ 4 ]
    selected for resources [ ]
    ===================================
    ps name: red-ps
    ps component: dtcp
    ps version: 1
    plugin: /lib/modules/4.1.10-irati/extra/red-plugin
    loaded for ipcps [ ]
    selected for resources [ ]
    ===================================
    ps name: default
    ps component: dtp
    ps version: 1
    plugin: /home/vmaffione/irati/local/lib/rinad/ipcp/rina-default-plugin
    loaded for ipcps [ 4 ]
    selected for resources [ ]
    ===================================
    ps name: default
    ps component: enrollment-task
    ps version: 1
    plugin: /home/vmaffione/irati/local/lib/rinad/ipcp/default
    loaded for ipcps [ 4 ]
    selected for resources [ 4 ]
    ===================================
    ps name: default
    ps component: flow-allocator
    ps version: 1
    plugin: /home/vmaffione/irati/local/lib/rinad/ipcp/default
    loaded for ipcps [ 4 ]
    selected for resources [ 4 ]
```

```
====================================
ps name: default
ps component: namespace-manager
ps version: 1
plugin: /home/vmaffione/irati/local/lib/rinad/ipcp/default
loaded for ipcps [ 4 ]
selected for resources [ 4 ]
====================================
ps name: default
ps component: pff
ps version: 1
plugin: /home/vmaffione/irati/local/lib/rinad/ipcp/rina-default-plugin
loaded for ipcps [ 4 ]
selected for resources [ 4 ]
====================================
ps name: lfa
ps component: pff
ps version: 1
plugin: /lib/modules/4.1.10-irati/extra/pff-lfa
loaded for ipcps [ ]
selected for resources [ ]
====================================
ps name: default
ps component: resource-allocator
ps version: 1
plugin: /home/vmaffione/irati/local/lib/rinad/ipcp/default
loaded for ipcps [ 4 ]
selected for resources [ 4 ]
====================================
ps name: cas-ps
ps component: rmt
ps version: 1
plugin: /lib/modules/4.1.10-irati/extra/cas-plugin
loaded for ipcps [ ]
selected for resources [ ]
====================================
ps name: default
ps component: rmt
ps version: 1
plugin: /home/vmaffione/irati/local/lib/rinad/ipcp/rina-default-plugin
loaded for ipcps [ 4 ]
selected for resources [ 4 ]
====================================
ps name: red-ps
ps component: rmt
ps version: 1
```

```
plugin: /lib/modules/4.1.10-irati/extra/red-plugin
loaded for ipcps [ ]
selected for resources [ ]
===================================
ps name: link-state
ps component: routing
ps version: 1
plugin: /home/vmaffione/irati/local/lib/rinad/ipcp/default
loaded for ipcps [ 4 ]
selected for resources [ 4 ]
===================================
ps name: default
ps component: sdup
ps version: 1
plugin: /home/vmaffione/irati/local/lib/rinad/ipcp/rina-default-plugin
loaded for ipcps [ 4 ]
selected for resources [ ]
===================================
ps name: PSOC_authentication-none
ps component: security-manager
ps version: 1
plugin: /home/vmaffione/irati/local/lib/rinad/ipcp/default
loaded for ipcps [ 4 ]
selected for resources [ ]
===================================
ps name: PSOC_authentication-password
ps component: security-manager
ps version: 1
plugin: /home/vmaffione/irati/local/lib/rinad/ipcp/default
loaded for ipcps [ 4 ]
selected for resources [ ]
===================================
ps name: PSOC_authentication-ssh2
ps component: security-manager
ps version: 1
plugin: /home/vmaffione/irati/local/lib/rinad/ipcp/default
loaded for ipcps [ 4 ]
selected for resources [ ]
===================================
ps name: default
ps component: security-manager
ps version: 1
plugin: /home/vmaffione/irati/local/lib/rinad/ipcp/default
loaded for ipcps [ 4 ]
selected for resources [ 4 ]
===================================
```

The `show-catalog` commands accepts the name of a component as an optional argument

```
IPCM >>> show-catalog pff
    ====================================
    ps name: default
    ps component: pff
    ps version: 1
    plugin: /home/vmaffione/irati/local/lib/rinad/ipcp/rina-default-plugin
    loaded for ipcps [ 4 ]
    selected for resources [ 4 ]
    ====================================
    ps name: lfa
    ps component: pff
    ps version: 1
    plugin: /lib/modules/4.1.10-irati/extra/pff-lfa
    loaded for ipcps [ ]
    selected for resources [ ]
    ====================================
```

In this case only the policy-sets for the specified component are shown.

The second command added is `update-catalog`, which can be used to trigger again the `import` operation described in Section 4.4.2 above. This is useful when new plugins are installed after the IPCM startup, and will cause the IPCM to update the database with the manifests of the new plugins. Existing database entries are not affected by this operation.

5. Plugins developer guide

The purpose of this chapter is to present some simple but complete examples that illustrate how to write user-space and kernel-space plugins. These tutorials are aligned to the second SDK iteration released with this deliverable.

5.1. User-space plugins tutorial

Although it is possible to develop user-space plugins within the *rinad/ipcp* source code tree, the preferred way of contributing custom plugins to the IPC Process user-space implementation is to develop them outside of the *rinad/ipcp* source tree. Consequently, this tutorial will focus on this case.

A user-space plugin contains the implementation of one or more policy-sets that will be made available to the IPC Process Daemon at runtime by dynamically linking the shared object containing the plugin. The recommended structure of a plugin consists of the following files, that are usually contained in a directory dedicated to the plugin:

- *plugin.cc*. The source file containing code that exports the policy-set factories for all the policy-sets contributed by the plugin.
- *<implementation>.cc*. A number of .cc and .h files containing the implementation of the policy-sets contributed by the plugin.
- *<plugin-name>.manifest*. A text file in JSON format that provides information on the policy-sets contributed by the plugin (see Section 4.3).
- *<makefiles>*. A build system to compile, link and install the plugin. For example, it may be a single Makefile, or more files for autoconf/automake ([autoconf],[automake]) or CMake.

In order to understand better how all pieces works together, this section will walk through an example of a simple plugin that contributes a single policy-set to the Security-Manager component of the IPCP Daemon. The plugin, which can be found in the *plugins/example-sm-plugin* directory of the [open-irati] repository, is made of the following files:

- *plugin.cc*
- *sm-example.cc*
- *sm-example.manifest*
- *Makefile*

5.1.1. Plugin.cc

```
#include <string>
#include <sstream>

#define IPCP_MODULE "security-manager-example-ps"
#include "ipcp-logging.h"
#include "components.h"    ❶

namespace rinad {

extern "C" rina::IPolicySet *
createSecurityManagerExamplePs(rina::ApplicationEntity * ctx);    ❷

extern "C" void
destroySecurityManagerExamplePs(rina::IPolicySet * ps);    ❸

extern "C" int
get_factories(std::vector<struct rina::PsFactory>& factories)    ❹
{
 struct rina::PsFactory factory;

 factory.info.name = "example";
 factory.info.app_entity =
 rina::ApplicationEntity::SECURITY_MANAGER_AE_NAME;
 factory.create = createSecurityManagerExamplePs;
 factory.destroy = destroySecurityManagerExamplePs;
 factories.push_back(factory);

 return 0;
}

}   // namespace rinad
```

❶ Inclusion of *components.h*, which contains the definitions of several IPC Process Daemon base classes and data structures. Among the others, there are the definitions of the base class for the Security Manager policy-sets, and the base class for all IPCP components.

❷ For each policy-set that the plugin contributes (one in this example), *plugin.cc* declares one function to create an instance of the policy-set.

❸ For each policy-set that the plugin contributes (one in this example), *plugin.cc* declares one function to destroy an instance of the policy-set.

❹ The definition of a `get_factories` function to export the policy-set factories. For each factory to be exported (in this case only one), a `rina::PsFactory` object instance is added to the output array passed as argument. Each entry in the array is built with the policy-set name, the IPC Process component (Application Entity) the policy set is part of (Security Manager in this case), and the references to the factory's *create* and *destroy* functions.

5.1.2. sm-example.cc

```
#include <string>
#include <sstream>

#define IPCP_MODULE "security-manager-example-ps"
#include "ipcp-logging.h"
#include "components.h"

namespace rinad {

class SecurityManagerExamplePs: public ISecurityManagerPs {    ❶
public:
 SecurityManagerExamplePs(IPCPSecurityManager * 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 ~SecurityManagerExamplePs() {}

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

 int max_retries;
};

// [...]

extern "C" rina::IPolicySet *
createSecurityManagerExamplePs(rina::ApplicationEntity * ctx)    ❷
{
 IPCPSecurityManager * sm = dynamic_cast<IPCPSecurityManager *>(ctx);

 if (!sm) {
  return NULL;
 }
```

```
 return new SecurityManagerExamplePs(sm);
}

extern "C" void
destroySecurityManagerExamplePs(rina::IPolicySet * ps)    ❸
{
 if (ps) {
  delete ps;
 }
}

}   // namespace rinad
```

❶ Definition of a class that inherits from the abstract class ISecurityManagerPS, which is the base class for all the Security Manager policy-sets. Such a class needs to provide an implementation for the *isAllowedToJoinDIF* and *acceptFlow* operations. We have omitted those dummy implementations from the listing since it is not important for the purpose of illustrating the SDK usage.

❷ Definition of the policy factory's create function, declared in *plugin.cc*. An IPCP component (or application entity) is passed as a parameter, and is downcasted to the object of type *IPCPSecurityManager*. Then a new instance of the *SecurityManagerExamplePs* class is created and returned to the caller.

❸ Definition of the destroy operation that deletes the instance of the policy set.

5.1.3. sm-example.manifest

```
{
        "PluginName": "sm-example",
        "PluginVersion": "1",
        "PolicySets" : [
                {
                        "Name": "example",
                        "Component": "security-manager",
                        "Version" : "1"
                }
        ]
}
```

The plugin manifest contains the name and version of the plugin as well as the name, component and version of all the policy-sets contributed by the plugin (just one in this case).

5.1.4. Makefile

```
# Set INSTALLDIR environment variable to the directory containing
# the IRATI installation

ifndef INSTALLDIR
$(error The INSTALLDIR environment variable is not set)
endif

PLUGIN_NAME = sm-example
SOURCES     =     \
 plugin.cc    \
 sm-example.cc

CXX        = g++
CXXFLAGS   = -Wall -Werror -fPIC
LDFLAGS    = -shared -L $(INSTALLDIR)/lib
LIBS       = -lrina

CXXFLAGS   += -I $(INSTALLDIR)/include/rinad/ipcp
CXXFLAGS   += -I $(INSTALLDIR)/include/rinad
CXXFLAGS   += -I $(INSTALLDIR)/include

OBJECTS    = $(SOURCES:.cc=.o)

all: $(PLUGIN_NAME).so

$(PLUGIN_NAME).so: $(OBJECTS)
 $(CXX) $(LDFLAGS) $(OBJECTS) -o $@

.cc.o:
 $(CXX) -c $(CXXFLAGS) $< -o $@

clean:
 rm -f $(OBJECTS)

install:
 install $(PLUGIN_NAME).so $(INSTALLDIR)/lib/rinad/ipcp
 install -m 0644 $(PLUGIN_NAME).manifest $(INSTALLDIR)/lib/rinad/ipcp

uninstall:
 -rm $(INSTALLDIR)/lib/rinad/ipcp/$(PLUGIN_NAME).so $(INSTALLDIR)/lib/
rinad/ipcp/$(PLUGIN_NAME).manifest
```

When invoking `make` in the plugin directory, the `INSTALLDIR` environment variable must be set to the absolute path of the IRATI installation (could be "/", if IRATI is installed system-wide). The `PLUGIN_NAME` variable contains the name of the plugin to be built, while the `SOURCES` variable contains the list of the C++ sources that are to be compiled and linked together in order to build the shared object.

5.2. Authentication plugins tutorial

Authentication is part of the Common Application Connection Establishment Phase (CACEP) that takes place between two IPCPs (and application processes in general) as illustrated in Figure 6. All the messages required for authentication are exchanged after the M_CONNECT message (which initiates the application connection setup procedure) and before the M_CONNECT_R message (which completes the application connection setup procedure).

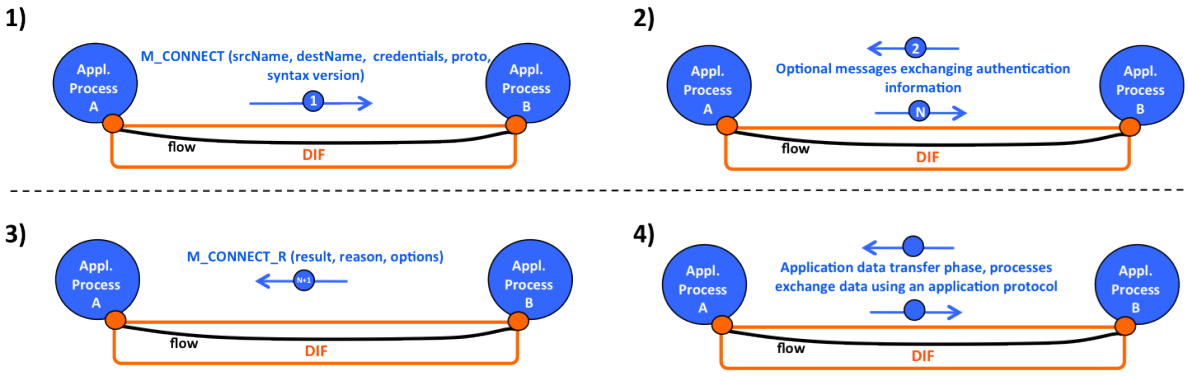

Figure 6. Authentication between APs when establishing an application connection

The messages exchanged during authentication belong to the authentication policy and can use any syntax that the authors of the policy consider appropriate. One of the potential options is to re-use the CDAP syntax, but without keeping the CDAP semantics. That is, authentication messages can re-use the message format defined in the CDAP specification (operation code, object name, object value, etc.), without interpreting the values of the message fields the same way as CDAP does (since the messages are just authentication exchanges and not operations on the RIB). As it will be seen later in the PoC implementation description, this approach simplifies the implementation since all the CDAP message parsing and generation machinery can be re-used.

5.2.1. APIs

When the IPC Process Daemon is created, it instantiates all the supported authentication policies and stores them in the Security Manager component by type. Each authentication policy must inherit from the *IAuthPolicySet* abstract class presented below.

```
class IAuthPolicySet : public IPolicySet {
public:
 enum AuthStatus {
  IN_PROGRESS, SUCCESSFULL, FAILED
 };

 IAuthPolicySet(const std::string& type_);
 virtual ~IAuthPolicySet() { };

 /// get auth_policy
 virtual cdap_rib::auth_policy_t get_auth_policy(int session_id,
           const AuthSDUProtectionProfile& profile) = 0;     ❶

 /// initiate the authentication of a remote AE. Any values originated
 /// from authentication such as sesion keys will be stored in the
 /// corresponding security context
 virtual AuthStatus initiate_authentication(const cdap_rib::auth_policy_t&
 auth_policy,
         const AuthSDUProtectionProfile& profile,
         int session_id) = 0;                                    ❷

 /// Process an incoming CDAP message
 virtual int process_incoming_message(const cdap::CDAPMessage& message,
          int session_id) = 0;                                              ❸

 //Called when the crypto state been updated on a certain port, if the
 call
 //to the Security Manager's "update crypto state" was asynchronous
 virtual AuthStatus crypto_state_updated(int port_id) = 0;
                        ❹

 // The type of authentication policy
 std::string type;
};
```

Each derived class has to override the following methods:

❶ **get_auth_policy**. Invoked by the RIB Daemon when it has to initiate an application connection with a remote application entity, in order to obtain the values for the *AuthPolicy* field of the CDAP M_CONNECT message. The policy gets the information on how to populate the *AuthPolicy* field from the security configuration of the IPC Process, which is passed as the *profile* argument of this method invocation.

❷ **initiate_authentication**. Invoked by the RIB Daemon when it receives an application connection request (CDAP M_CONNECT message) from a remote application entity. This operation returns *SUCCESS* if authentication is successful, *FAILURE* if it fail or *IN PROGRESS* if more messages need to be exchanged.

❸ **process_incoming_message**. Invoked by the RIB Daemon when it receives an authentication-related message. Return type is the same than the former operation.

❹ **crypto_state_updated**. Callback informing about the result of an "update crypto state" call to the Security Manager, in case this operation is asynchronous (as it is the case of the IPC Process, which involves sending a Netlink message to the kernel and getting the response back asynchronously).

5.2.2. Example: the AuthNPassword policy

Figure 7 illustrates the workflow of this authentication policy. It is based on a pre-shared password that both parties need to obtain before authenticating. The same password could be shared by all DIF members, or different passwords could be used. IPCP A populates the *'AuthPolicy'* field with the following data:

- **Name**: PSOC_authentication-password.
- **Version**: 1 (only supported version as of now).
- **Options**: empty.

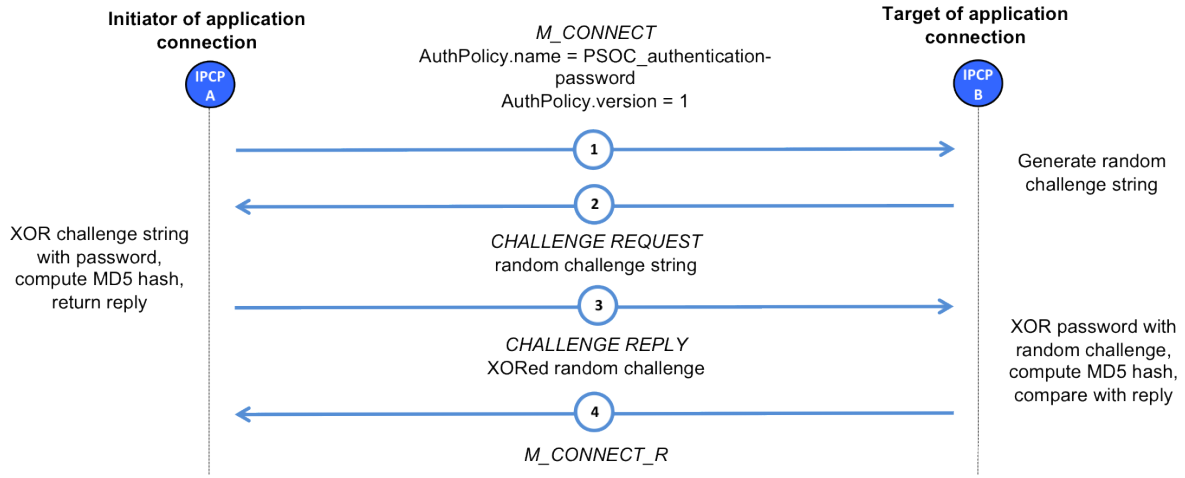

Figure 7. Workflow of AuthNPassword policy

Upon receiving the M_CONNECT message, IPCP B decides if the authentication policy is appropriate. If it is, it generates a random string of a certain length (which has to match the password length in order not to weaken the strength of the authentication, based on XORing the password with the random string). Once the string is generated, IPCP B creates a CDAP M_WRITE message with the information below, and sends it to IPCP A.

- **Opcode**: M_WRITE.
- **Object class**: challenge request.
- **Object value**: <type> = string, <value> = <the random string generated by IPCP B>.

Once IPCP A receives the message, it XORs the random string with the password, computes the MD5 hash of the result and sends the hashed value back to IPCP B in the following message.

- **Opcode**: M_WRITE.
- **Object class**: challenge reply.
- **Object value**: <type> = string, <value> = <random string XORed with password>.

Once IPCP B receives the message, it XORs the random challenge with the password, applies the MD5 hash and compares the result with the value received from IPCP A. If the values are the same, the authentication is successful and the IPCP invokes the DIF/DAF access control policy (which will end up sending an M_CONNECT_R message back to IPCP A if successful). If not, authentication fails and IPCP B sends an M_RELEASE CDAP message back to IPCP A.

5.2.2.1. Implementation

This policy is implemented by the class *AuthPasswordPolicySet* which extends the abstract class *IAuthPolicySet*. According to the workflow depicted on Figure 7, the first operation that is invoked by IPCP A is **get_auth_policy**.

```
cdap_rib::auth_policy_t AuthPasswordPolicySet::get_auth_policy(int
 session_id,
                     const AuthSDUProtectionProfile& profile)
{
 if (profile.authPolicy.name_ != type) {  ❶
  LOG_ERR("Wrong policy name: %s",
                          profile.authPolicy.name_.c_str());
  throw Exception();
 }

 AuthPasswordSecurityContext * sc =
                         new AuthPasswordSecurityContext(session_id);  ❷
 sc->password = profile.authPolicy.get_param_value_as_string(PASSWORD);
 if (sc->password == std::string()) {
  LOG_ERR("Could not find password parameter");
  throw Exception();
 }
 sc->challenge_length = sc->password.length();
 sc->crcPolicy = profile.crcPolicy;
```

```
 sc->ttlPolicy = profile.ttlPolicy;
 sc->con.port_id = session_id;
 sec_man->add_security_context(sc);  ❸

 cdap_rib::auth_policy_t result;
 result.name = IAuthPolicySet::AUTH_PASSWORD;
 result.versions.push_back(profile.authPolicy.version_);
 return result;  ❹
}
```

❶ The function first checks the configuration to make sure this is the policy that should be executing.

❷ A security context associated to the authentication session id is created, in order to store all the information that will be needed while authentication takes place.

❸ The function gets the password and other security-related data from the IPC Process security configuration, and asks the IPCP Security Manager to store the security context for the authentication session.

❹ Finally the function generates the aut_policy_t data structure and returns it to the RIB Daemon, who will send an M_CONNECT message to B in order to initiate the application connection establishment.

When IPCP B gets the M_CONNECT CDAP message, it will pass it to the authentication policy, invoking the **initiate_authentication** operation.

```
rina::IAuthPolicySet::AuthStatus
 AuthPasswordPolicySet::initiate_authentication(const AuthPolicy&
 auth_policy,
          const AuthSDUProtectionProfile& profile,
                      int session_id)
{
 if (auth_policy.name_ != type) {  ❶
  LOG_ERR("Wrong policy name: %s", auth_policy.name_.c_str());
  return rina::IAuthPolicySet::FAILED;
 }

 if (auth_policy.versions_.front() != RINA_DEFAULT_POLICY_VERSION) {
  LOG_ERR("Unsupported policy version: %s",
    auth_policy.versions_.front().c_str());
  return rina::IAuthPolicySet::FAILED;
 }

 AuthPasswordSecurityContext * sc = new
 AuthPasswordSecurityContext(session_id);  ❷
```

```
sc->password = profile.authPolicy.get_param_value_as_string(PASSWORD);
if (sc->password == std::string()) {
 LOG_ERR("Could not find password parameter");
 throw Exception();
}
sc->challenge_length = sc->password.length();
sc->crcPolicy = profile.crcPolicy;
sc->ttlPolicy = profile.ttlPolicy;
sec_man->add_security_context(sc);

ScopedLock scopedLock(lock);

//1 Generate a random password string and send it to the AP being
authenticated
sc->challenge = generate_random_challenge(sc->challenge_length);  ❸

try {
 RIBObjectValue robject_value;
 robject_value.type_ = RIBObjectValue::stringtype;
 robject_value.string_value_ = *(sc->challenge);

 RemoteProcessId remote_id;
 remote_id.port_id_ = session_id;

 //object class contains challenge request or reply
 //object name contains cipher name
 rib_daemon->remoteWriteObject(CHALLENGE_REQUEST, CHALLENGE_REQUEST,  ❹
   robject_value, 0, remote_id, 0);
} catch (Exception &e) {
 LOG_ERR("Problems encoding and sending CDAP message: %s", e.what());
}

//2 set timer to clean up pending authentication session upon timer
expiry
sc->timer_task = new CancelAuthTimerTask(sec_man, session_id);
timer.scheduleTask(sc->timer_task, timeout);

return rina::IAuthPolicySet::IN_PROGRESS;
          ❺
}
```

❶ Check that the requested authentication policy name and version are consistent with those of the authentication policy that has been called.

❷ Create a security context for the session, get password from configuration and store it in the security context.

❸ Generate a random challenge string.

❹ Encode the string as an object value of a write operation, and invoke the operation on the remote peer (will cause CDAP M_WRITE message to be sent to the peer).

❺ Return a value to indicate that the authentication is still in progress.

Now IPCP B has sent a CDAP M_WRITE message to IPCP A with the *Challenge request* object name, containing the random challenge string. IPCP A will receive the message and pass it to the authentication policy using the the **process_incoming_message** operation.

```
int AuthPasswordPolicySet::process_incoming_message(const CDAPMessage&
 message,
           int session_id)
{
 StringObjectValue * string_value = 0;
 const std::string * challenge = 0;

 if (message.op_code_ != CDAPMessage::M_WRITE) {  ❶
  LOG_ERR("Wrong operation type");
  return IAuthPolicySet::FAILED;
 }

 if (message.obj_value_ == 0) {  ❷
  LOG_ERR("Null object value");
  return IAuthPolicySet::FAILED;
 }

 string_value = dynamic_cast<StringObjectValue *>(message.obj_value_);
 if (!string_value) {
  LOG_ERR("Object value of wrong type");
  return IAuthPolicySet::FAILED;
 }

 challenge = (const std::string *) string_value->get_value();  ❸
 if (!challenge) {
  LOG_ERR("Error decoding challenge");
  return IAuthPolicySet::FAILED;
 }

 if (message.obj_class_ == CHALLENGE_REQUEST) {  ❹
  return process_challenge_request(*challenge,
       session_id);
 }

 if (message.obj_class_ == CHALLENGE_REPLY) {  ❺
```

```
  return process_challenge_reply(*challenge,
            session_id);
 }

 LOG_ERR("Wrong message type");
 return IAuthPolicySet::FAILED;
}
```

❶ Check that the message is of type M_WRITE (only these messages are used by this policy).

❷ Check that the object value is not null.

❸ Recover the object value.

❹ If the message is of type *Challenge request* invoke the function to process it.

❺ If the message is of type *Challenge response* invoke the function to process it.

Since the message is of type *Challenge request*, the following function will be invoked:

```
int AuthPasswordPolicySet::process_challenge_request(const std::string&
 challenge,
           int session_id)
{
 ScopedLock scopedLock(lock);

 AuthPasswordSecurityContext * sc =
   dynamic_cast<AuthPasswordSecurityContext *>(sec_man-
>get_security_context(session_id));  ❶
 if (!sc) {
  LOG_ERR("Could not find pending security context for session_id %d",
   session_id);
  return IAuthPolicySet::FAILED;
 }

 try {
  RIBObjectValue robject_value;
  robject_value.type_ = RIBObjectValue::stringtype;
  robject_value.string_value_ = encrypt_challenge(challenge, sc-
>password);  ❷

  RemoteProcessId remote_id;
  remote_id.port_id_ = session_id;

  //object class contains challenge request or reply
  //object name contains cipher name
  rib_daemon->remoteWriteObject(CHALLENGE_REPLY, CHALLENGE_REPLY,  ❸
    robject_value, 0, remote_id, 0);
 } catch (Exception &e) {
```

```
  LOG_ERR("Problems encoding and sending CDAP message: %s", e.what());
 }

 return IAuthPolicySet::IN_PROGRESS;  ❹
}
```

❶ Recover the security context for this session.

❷ XOR the random challenge with the password, and hash it.

❸ Send the result encoded as an string, invoking a remote write operation on the peer IPCP (will cause a CDAP M_WRITE message to be sent to the peer IPCP).

❹ Return an *authentication in progress* result to the caller.

When this function completes a CDAP M_WRITE message with the *Challenge reply* object name will be send to IPCP B. IPCP B will receive it, and invoke the *process cdap message* operation of the authentication policy. The policy will invoke the **process challenge reply** operation, described below.

```
int AuthPasswordPolicySet::process_challenge_reply(const std::string&
 encrypted_challenge,
         int session_id)
{
 int result = IAuthPolicySet::FAILED;

 ScopedLock scopedLock(lock);

 AuthPasswordSecurityContext * sc =
   dynamic_cast<AuthPasswordSecurityContext *>(sec_man-
>get_security_context(session_id));  ❶
 if (!sc) {
  LOG_ERR("Could not find pending security context for session_id %d",
   session_id);
  return IAuthPolicySet::FAILED;
 }

 timer.cancelTask(sc->timer_task);

 std::string recovered_challenge = decrypt_challenge(encrypted_challenge,
 sc->password);  ❷
 if (*(sc->challenge) == recovered_challenge) {  ❸
  result = IAuthPolicySet::SUCCESSFULL;
 } else {
  LOG_DBG("Authentication failed");
 }
```

```
  return result;
}
```

❶ Recover the security context for this session.

❷ XOR the original challenge with the password and hash the result.

❸ Compare the result of the operation with the received challenge, if they are equal authentication is successful, otherwise authentication fails.

Once the authentication policy returns success/failure, the IPCP Daemon code in IPCP B will send an M_CONNECT response message to IPCP A, indicating about the success or failure in establishing the application connection.

5.3. Kernel-space plugins tutorial

Custom plugins for the kernel-space part of the IPC Process are developed as out-of-tree loadable Linux kernel modules. A kernel-space plugin contains the implementation of one or more policy-sets that can be published to the kernel-space SDK at runtime by dynamically loading the associated kernel module.

The recommended structure of a kernel-space plugin contains the following files, that are usually put in a same directory dedicated to the plugin:

- *plugin.c*. The source file that contains the module `init()` and `exit()` functions that are in charge of respectively publishing and un-publishing the policy-set factories for all the policy-sets contributed by the plugin.
- *<implementation>.c*. A number of .c and .h files containing the implementation of the policy-sets hooks (methods), and the implementation of the constructor and destructor factory callbacks.
- *<plugin-name>.manifest*. A text file in JSON format that provides information about the plugin itself and the list of the policy-sets contributed by the plugin (see section Section 4.3).
- *Makefile*. A makefile to build the plugin as an out-of-tree kernel module.

5.3.1. The Loop Free Alternates plugin

Similarly to what presented in Section 5.1, this section will walk through a concrete example of a simple plugin that contributes a single policy-set to the PDU Forwarding Function (PFF) component of the IPCP Daemon. The plugin, which can be found in the *plugins/lfa* directory of [open-irati-stack], is made up of the following files:

- *plugin.c*
- *lfa-ps.c*

- *pff-lfa.manifest*
- *Makefile*

5.3.2. plugin.c

```
#include <linux/export.h>
#include <linux/module.h>
#include <linux/string.h>

#define RINA_PREFIX "pff-lfa"

#include "logs.h"
#include "rds/rmem.h"
#include "rds/rtimer.h"
#include "pff-ps.h"    ❶
#include "debug.h"

#define RINA_PFF_LFA_NAME "lfa"

extern struct ps_factory pff_factory;    ❷

static int __init mod_init(void)
{
 int ret;

 strcpy(pff_factory.name, RINA_PFF_LFA_NAME);

 ret = pff_ps_publish(&pff_factory);    ❸
 if (ret) {
  LOG_ERR("Failed to publish policy set factory");
  return -1;
 }

 LOG_INFO("PFF LFA policy set loaded successfully");

 return 0;
}

static void __exit mod_exit(void)
{
 int ret;

 ret = pff_ps_unpublish(RINA_PFF_LFA_NAME);    ❹
```

```
 if (ret) {
  LOG_ERR("Failed to unpublish policy set factory");
  return;
 }

 LOG_INFO("PFF LFA policy set unloaded successfully");
}

module_init(mod_init);
module_exit(mod_exit);

MODULE_DESCRIPTION("PFF LFA policy set");
MODULE_LICENSE("GPL");
```

❶ Inclusion of *pff-ps.h*, which contains the declaration of `struct pff_ps`, that is the base class for all PFF policy-sets.

❷ For each policy-set that the plugin contributes (one in this example), *plugin.c* declares the corresponding policy-set factory, which is defined in one of the implementation files, in this case *lfa-ps.c*.

❸ In the module loading function, the plugin publishes all the factories to the kernel-space SDK. In this case, there is only one policy-set, for the PFF component, and so the PFF-specific publishing routine is used.

❹ In the module unloading function, the plugin un-publishes all the factories published to the kernel-space SDK (only one in this case).

5.3.3. lfa-ps.c

```
#include <linux/export.h>
#include <linux/module.h>
#include <linux/string.h>

#define RINA_PREFIX "pff-lfa"

#include "logs.h"
#include "rds/rmem.h"
#include "rds/rtimer.h"
#include "pff-ps.h"
#include "debug.h"

struct pft_port_entry {
 port_id_t port_id;
 struct list_head next;
};
```

```
struct pft_entry {
 address_t destination;
 qos_id_t qos_id;
 port_id_t nhop;
 port_id_t port;
 struct list_head alt_ports;
 struct list_head next;
};

/* Policy-set-specific data structure to be shared among the
 * PFF policies. */
struct pff_ps_priv {    ❶
 spinlock_t lock;
 struct list_head entries;
 /* Holds ports that are down */
 struct list_head ports_down;
};

/* Policy invoked from the IRATI stack to add an entry to the LFA PFF. */
static int lfa_add(struct pff_ps *ps,    ❷
                   struct mod_pff_entry *entry)
{
        struct pff_ps_priv *priv;
        struct pft_entry *tmp;
        struct port_id_altlist *alts;
        int i;

        priv = (struct pff_ps_priv *) ps->priv;
        if (!priv_is_ok(priv))
                return -1;

        if (!entry) {
                LOG_ERR("Bogus output parameters, won't add");
                return -1;
        }

        if (!is_address_ok(entry->fwd_info)) {
                LOG_ERR("Bogus destination address passed, cannot add");
                return -1;
        }
        if (!is_qos_id_ok(entry->qos_id)) {
                LOG_ERR("Bogus qos-id passed, cannot add");
                return -1;
        }
```

```
        spin_lock(&priv->lock);

        tmp = pft_find(priv, entry->fwd_info, entry->qos_id);
        if (!tmp) {
                tmp = pfte_create_ni(entry->fwd_info, entry->qos_id);
                if (!tmp) {
                        spin_unlock(&priv->lock);
                        return -1;
                }

                list_add(&tmp->next, &priv->entries);
        } else {
                LOG_ERR("Entry already exists");
                spin_unlock(&priv->lock);
                return -1;
        }
        alts = list_first_entry(&entry->port_id_altlists,
                                typeof(*alts),
                                next);

        if (alts->num_ports < 1) {
                LOG_INFO("Port id set is empty");
                pfte_destroy(tmp);
                spin_unlock(&priv->lock);
                return -1;
        }

        tmp->port = alts->ports[0];
        tmp->nhop = tmp->port;

        for (i = 1; i < alts->num_ports; i++) {
                if (pfte_port_add(tmp, alts->ports[i])) {
                        pfte_destroy(tmp);
                        spin_unlock(&priv->lock);
                        return -1;
                }
        }

        spin_unlock(&priv->lock);

        return 0;
}

// [...]

/* Constructor for LFA policy-set instances. */
```

```
static struct ps_base *
pff_ps_lfa_create(struct rina_component *component)   ❸
{
 struct pff_ps *ps;
 struct pff_ps_priv *priv;
 struct pff *pff = pff_from_component(component);

        /* Allocate memory for the shared data structure. */
 priv = rkzalloc(sizeof(*priv), GFP_KERNEL);
 if (!priv)
  return NULL;

        /* Initialized the shared data structure. */
 spin_lock_init(&priv->lock);
 INIT_LIST_HEAD(&priv->entries);
 INIT_LIST_HEAD(&priv->ports_down);

        /* Allocate memory for the policy-set instance. */
 ps = rkzalloc(sizeof(*ps), GFP_KERNEL);
 if (!ps)
  return NULL;

        /* Initialize references to data model and shared data
         * structure. */
 ps->base.set_policy_set_param = NULL; /* default */
 ps->dm = pff;
 ps->priv = (void *) priv;

        /* Fill in the instance's hooks. */
 ps->pff_add = lfa_add;
 ps->pff_remove = lfa_remove;
 ps->pff_port_state_change = lfa_port_state_change;
 ps->pff_is_empty = lfa_is_empty;
 ps->pff_flush = lfa_flush;
 ps->pff_nhop = lfa_nhop;
 ps->pff_dump = lfa_dump;

 return &ps->base;
}

/* Destructor for LFA policy-set instances. */
static void pff_ps_lfa_destroy(struct ps_base *bps)   ❹
{
 struct pff_ps *ps = container_of(bps, struct pff_ps, base);

 if (bps) {
```

```
    struct pff_ps_priv *priv;

    priv = (struct pff_ps_priv *) ps->priv;
    if (!priv_is_ok(priv))
     return;

                /* Clean up the shared data structure. */
    spin_lock(&priv->lock);

    __pft_flush(priv);

    spin_unlock(&priv->lock);

                /* Deallocate the shared data structure and the
                 * policy-set instance. */
    rkfree(priv);
    rkfree(ps);
   }
}

struct ps_factory pff_factory = {    ❺
 .owner   = THIS_MODULE,
 .create  = pff_ps_lfa_create,
 .destroy = pff_ps_lfa_destroy,
};
```

❶ The definition of the LFA-specific private data structure for the LFA policy-set, used to implement shared state among the policies implementation. For the LFA plugin, the private data structure contains a list of forwarding entries (i.e. the forwarding table), the list of ports that are currently down, and a lock for concurrent access.

❷ Definition of the `lfa_add` policy, invoked by the IRATI stack to add an entry to the PDU Forwarding Table. As explained above, a pointer to this function is used to initialize the policy-set hooks. The other hooks are not shown in the listing above, since it's not the purpose of this tutorial to explain the LFA algorithm and data structures.

❸ The definition of the of the factory's create function, to be exported to the SDK. The function takes a pointer to the `struct rina_component` base class, which is actually pointing to an object of derived type `struct pff`. It allocates memory for either the policy-set instance and the private data structure. The `priv` and `dm` fields of the policy-set instance are properly initialized with references to the associated private data structure and data model, and the hooks are filled with pointers to locally defined functions that implement single policies (e.g. `lfa_add` (2), `lfa_remove`, etc.). Finally, the address of the policy-set instance just created is returned.

❹ The definition of the factory's destroy function, to be exported to the SDK. This function flushes (deallocates) the entries in the forwarding table, deallocates the memory of the policy-set instance and of the private data structure.

❺ The definition of the LFA factory, as a global variable in the kernel module. The `owner` field must be set to `THIS_MODULE` to correctly support reference counting, so that the module cannot be unloaded while LFA policy-set instances are being used. The `create` and `destroy` factory callbacks are initialised with the locally defined functions described above.

5.3.4. pff-lfa.manifest

```
{
        "PluginName": "pff-lfa",
        "PluginVersion": "1",
        "PolicySets" : [
                {
                        "Name": "lfa",
                        "Component": "pff",
                        "Version" : "1"
                }
        ]
}
```

The plugin manifest contains the name and version of the plugin as well as the name, component and version of the LFA PFF policy-set.

5.3.5. Makefile

```
KDIR=../../linux
KREL=`uname -r`

ifneq ($(KERNELRELEASE),)

ccflags-y = -Wtype-limits -Inet/rina

obj-m := pff-lfa.o    ❶

pff-lfa-y := lfa-ps.o plugin.o    ❷

else

all:
```

```
 $(MAKE) -C $(KDIR) M=$$PWD    ❸

clean:
 rm -rf *.o *.ko *.mod.c *.mod.o

install:    ❹
 $(MAKE) -C $(KDIR) M=$$PWD modules_install
 cp pff-lfa.manifest /lib/modules/$(KREL)/extra/

endif
```

❶ defines a new module called `pff-lfa`,

❷ specifies what compilation units the `pff-lfa` module is made of.

❸ invokes the Linux build system to compile and link the module, while

❹ installs the built module, together with the manifest file, in a system directory dedicated to out-of-tree modules.

The Makefile is written to be integrated into the Linux build system.

6. Updates to high-level programming languages bindings

An overview of how the binding process works has been presented in section 6 of [pristine-d23], both the concepts and a worked binding example. This section presents only the updated aspects of the Java bindings for [swig]. A number of changes were necessary to improve the structure of the generated bindings and to optimise the amount of generated classes, for example, removing unnecessary implementation details from the header files, so as the generated wrapping classes are smaller.

6.1. Separation of the librina.i and librinad.i

In order to keep the number of generated classes to a minimum it was necessary to separate some of the original `librina.i` directives into two files. This allows some applications to use the application aspects e.g. registration and unregistration, flow allocation etc., without including CDAP encoding aspects. This has the result that some applications are smaller, as they include only the bindings they need.

However, there is a slight complication in that there needs to be a dependency with the `librina` wrapped classes. This dependency avoids a problem where the same class is wrapped twice (once for each file) leading to conflicts at compile time. To facilitate this separation a "wrap" folder has been added to the `rinad` folder with an equivalent folder layout as that of the "wrap" folder in `librina`.

Thus the CDAP encoder interface was moved from the general `librina` wrappings to specific CDAP one (`librinad`). There are a number of improvements in the header file structure, to separate the data-types from the CDAP encoder definitions. This change facilitates easier wrapping of the headers.

Finally, both sets of generated jars now have proper name-spacing applied (`eu.irati.librina` and `eu.irati.librinad` respectively).

6.2. librinad.i changes

This section gives an overview of the directives in the `librinad.i`.

The first directive imports the `ser_obj_t` type which is generated from `librina.i`. This prevents duplication of the type definition in the generated JNI native code.

```
%pragma(java) jniclassimports=%{
import eu.irati.librina.ser_obj_t;
```

```
%}
```

There is an equivalent complimentary adjustment to the Java imports (for the generated Java wrapping classes) later in the file.

```
%typemap(javaimports) SWIGTYPE
%{
import eu.irati.librina.ser_obj_t;
%}
```

The following directives include the relevant CPP header files into the JNI code. The final directive includes the Management Agent (MA) type structures into the generated mappings.

```
%{
#include "structures_mad.h"
#include "librina/cdap_rib_structures.h"
#include "encoders_mad.h"
#include <string>
%}

%include "structures_mad.h"
```

With the basic type imports completed. The next set of directives instantiates some template classes for encoding `std:string`, `ipcp_config_t` and `ipcp_t` types. Once these template types are available we can wrap the equivalent encoders defined in "encoders_mad" header file namely `StringEncoder`, `IPCPConfigEncoder` and `IPCPEncoder`.

```
namespace rinad{
namespace mad_manager{
%template(TempStringEncoder) Encoder<std::string>;
%template(TempIPCPConfigEncoder) Encoder<ipcp_config_t>;
%template(TempIPCPEncoder) Encoder<ipcp_t>;
}}

%include "encoders_mad.h"
```

The next section defines a macro for allowing iterators to be added to a C++ collection class. First it defines an implementation of the standard Java iterator interface for the equivalent wrapped type.

```
/* 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();
  }
%}
```

It adds one private method to the JNI class called `nextImpl`. This is declared as an inline method of the generated JNI class. The implementation instantiates an iterator for the underlying C++ collection.

```
%javamethodmodifiers ITERATORNAME::nextImpl "private"; ❶
%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>" ❷
%newobject CPPCOLLECTION<CPPTYPE>::iterator() const;
%extend CPPCOLLECTION<CPPTYPE> {  ❸
  ITERATORNAME *iterator() const {
    return new ITERATORNAME(*$self);
```

```
  }
}
%enddef
```

❶ is the JNI implementation of appropriate iterator class.

❷ ensures the generated Java wrappers implement the `Iterable` interface.

❸ extends the generated JNI wrapper to have a method called `iterator` that creates the appropriate iterator class.

The final section instantiates a Java type `StringList` for the template class `std::list` containing a `std::string`.

```
%template(StringList) std::list<std::string>;
```

6.2.1. librina.i changes

There are a number of changes to the `librina.i`. These can be summarised as follows:

1. Removal of the CDAP encoder related definitions.
2. Use of directors to allow modifiable callback behaviour.
3. Bug fixes on the bindings related to PR775[1].

The movement of the CDAP encoder from `librina.i` to `librinad.i` has already been discussed. However, the other changes are discussed below.

6.2.1.1. Use of directors for callbacks

One of the major improvement is to use directors to allow native C++ code to call Java implementation classes. The correct behaviour for example, when a new flow is established is application process specific, thus the method that is called has to be overloaded (in Java) with the appropriate Java implementation code.

The following snippet demonstrates the use of the director feature on the `CDAPCallbackInterface` class.

```
module(directors="1") cdapcallbackjava ❶

%{
```

[1] https://github.com/IRATI/stack/pull/775

```
#include <iostream>
#include "librina/exceptions.h"
#include "librina/patterns.h"
#include "librina/concurrency.h"
#include "librina/common.h"
#include "librina/application.h"
#include "librina/cdap_rib_structures.h"
#include "librina/cdap_v2.h"
#include "librina/ipc-api.h"
%}

%feature("director") rina::cdap::CDAPCallbackInterface; ❷
```

❶ enables the use of directors in the wrapping process

❷ instructs SWIG to use the directors on the C++ class `CDAPCallbackInterface`

6.2.1.2. Bug fixes

There are a number of bug fixes applied to the bindings to eliminate undesireable behaviour

The first of the bug fixes was to ensure that the `APPLICATION_UNREGISTERED_EVENT` and `APPLICATION_REGISTRATION_CANCELED_EVENT` were propagated in the wrapper classes to the Java application. The default behaviour was to omit passing on these events. This allows the Java to preform any necessary clean up operations. For brevity the implementation for `APPLICATION_UNREGISTERED_EVENT` is only shown below.

```
[...]
} else if ($1->eventType == rina::APPLICATION_UNREGISTERED_EVENT) {
  rina::ApplicationUnregisteredEvent *appUnregisteredEvent
 = dynamic_cast<rina::ApplicationUnregisteredEvent *>($1);
  jclass clazz = jenv->FindClass("eu/irati/librina/
ApplicationUnregisteredEvent");
  if (clazz) { ❶
    jmethodID mid = jenv->GetMethodID(clazz, "<init>", "(JZ)V");
    if (mid) {
      jlong cptr = 0;
      *(rina::ApplicationUnregisteredEvent **)&cptr =
 appUnregisteredEvent;
      $result = jenv->NewObject(clazz, mid, cptr, false); ❷
    }
  }
```

❶ finds the appropriate class for the event.

❷ instantiates a Java object for the event

A second bug was a clean up of the Exception throwing code, to reduce the possibility of the C ++ exception wrapping code generating a exception where there is no corresponding Java class (in the generated wrappers). This possibility exists as not all internal exceptions (in librina and rinad) are processed by SWIG. Currently, only the subset of those exceptions that should occur on the IPC API application interface are processed.

Further bug-fixes (e.g. `NullPointerException`) have also been done to improve overall stability.

7. Conclusions and future work

This document presented the consolidated version of the PRISTINE Software Development Kit, which is one of the fundamental outputs of the PRISTINE project. The overall software architecture has not changed considerably with reference to the first SDK version, and differences have been reported here. As a consequence, D2.3 is still a valid reference.

Considerable effort has been put into broadening the SDK usability, automation, and coverage of component and policies. Tutorials have been included to enrich the overall SDK documentation. D2.3 and D2.5 are therefore a complete reference for the RINA programmability offered by the IRATI open source prototype. The SDK-enhanced IRATI software is available at [open-irati], in the pristine-1.3 branch.

PRISTINE's Description of Work (DoW) document describes a number of metrics which capture the expectations on the SDK's features and completeness. These metrics are summarized in the following table.

Table 1. PRISTINE SDK evaluation metrics, as presented in the "Description of Work"

No	Metric	Description
3	Coverage of the PRISTINE SDK	This indicator will measure the "code coverage of the SDK", that is, of all the functionality required to implement the policies specified by WPs 3-5, which has been developed using the SDK and which has required a source code modification (because the SDK still doesn't cover it).
6	Expressions of interest in the usage of the PRISTINE SDK, by organizations external to the project	This indicator will showcase the potential impact that the SDK developed by PRISTINE can achieve. Since the full SDK will not be available until the final stages of the project, it is difficult that a lot of organizations external to the project can actually use it. However the consortium will promote the SDK and allow limited access to a number of interested parties, starting by the members of the PRISTINE External Advisory Board. The PRISTINE consortium expects to involve at least 3 academic and 2 industrial organizations (including SMEs)

No	Metric	Description
		in the usage of the SDK during the project lifetime.

Regarding PRISTINE metric number 3, all the policies developed up to now in the project are SDK plugins, therefore this metric has been 100% achieved (however it should be re-evaluated at the end of the project, after completing the development stage of phase 2). The table below captures the policy-sets developed by WPs 3-5 to date.

Table 2. PRISTINE policies developed to date

Name, description	Research area	SDK policy-sets
Congestion avoidance (CAS). Raj Jain's binary feedback scheme for congestion avoidance	Congestion management	RMT, DTCP
RED + TCP. Random Early Detection using ECN-marking and TCP's congestion avoidance scheme	Congestion management	RMT, DTCP
Cherish-urgency. Delta-t's cherish-urgency multiplexing technique for scheduling PDUs	Resource allocation	RMT
Simple multi-path forwarding. ECMP-like PDU Forwarding Function	Resource Allocation	Routing, PDU Forwarding Function (PFF)
LFA. Loop-Free Alternate routing	Resiliency	Routing, PDU Forwarding Function (PFF)
AuthPassword. Authentication policy based on a hashed shared password	Authentication	CACEP
AuthSSH2. Authentication policy based on SSH2 (based on shared RSA keys)	Authentication	CACEP
Defaultcrypto. Default cryptographic SDU protection which uses AES(128 or 256) to encrypt in block mode	Confidentiality	SDU Protection - cryptographic protection
CRC32. An SDU Protection policy that uses a CRC32 to detect errors in the PDU	Resiliency	SDU Protection - error check

Name, description	Research area	SDK policy-sets
TTL. A PDU lifetime termination enforcement mechanism based on a maximum number of hops	Resiliency	SDU Protection - lifetime limiting

Regarding PRISTINE metric number 6, it is now when the SDK is becoming complete and stable enough for external parties to start using it, therefore this metric must be evaluated at the end of the project. As described in the DoW, PRISTINE will increase the dissemination activities related to promoting the SDK, in order to attract users external to the project. One of the most important dissemination actions carried out by the project in this regard - actually suggested by PRISTINE reviewers - is to provide tutorials that allow researchers to reproduce the experiments described in papers submitted by PRISTINE (the tutorials are available online at [open-irati-tutorials]).

Although task T2.3 ends with this deliverable, SDK-related bug-fixing and code hardening activities will continue in the context of WP6, to properly support experimentation activities and further feedbacks from SDK users.

References

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

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

[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

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

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

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

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

[open-irati-tutorials] Tutorials to reproduce different RINA experiments with the IRATI implementation. Available online at https://github.com/IRATI/stack/wiki/Tutorials

[pristine-d23] PRISTINE project. 'D2.3 - Proof of concept of the Software Development Kit'. Available online at http://ict-pristine.eu/wp-content/uploads/2013/12/pristine-d23-sdk-v1_0.pdf

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