Deliverable-4.2 (1st version)

Initial Specification and Proof of Concept Implementation of Innovative Security and Reliability Enablers

Deliverable Editor: Hamid Asgari, Thales UK Research & Technology

Publication date:	08-June-2015
Deliverable Nature:	Report
Dissemination level (Confidentiality):	PU (Public)
Project acronym:	PRISTINE
Project full title:	PRogrammability In RINA for European Supremacy of virTualIsed NEtworks
Website:	www.ict-pristine.eu
Keywords:	Security, DIF, DAF, IPC Process, access control, authentication, SDU protection, resiliency
Synopsis:	D4.2 describes the initial specifications and proof of concept implementations of the security functions and enablers developed within WP4 to enable networks that are more secure and reliable than those we have today.

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: Hamid Asgari, Thales UK Research & Technology
TRT: Sarah Haines, Hamid Asgari
FIT-BUT: Ondrej Rysavy, Ondrej Lichtner
i2CAT: Eduard Grasa
iMINDS: Sander Vrijders, Dimitri Staessens
IMT: Fatma Hrizi, Anis Laouiti, Hakima Chaouchi
TSSG: Ehsan Elahi, Micheal Crotty, Jason Barron
NXW: Vincenzo Maffione

Disclaimer

Executive Summary

The security objective is set to reduce the security risks as much as possible by defining security functions and enablers. This document, D4.2, builds upon the security functions, mechanisms, and techniques that are described in D4.1 [D4.1] and provides their further developments within WP4 to meet the requirements enabling more secure and reliable networks than those that we have today. These functions, mechanisms and techniques include the Authentication, Access Controls (Capability-Based Access Control and Multi-Level Security) Cryptographic function, Key Management and Resiliency aspects of security. The deliverable overall provides the relevant specifications and analysis, the design aspects, Proof of Concept implementations (PoC), and related PoC tests.

Given the guidelines stated in the introduction section of this deliverable, therefore, in the following sections of the deliverable we provide, to a certain extent, the description of the following aspects in relation to all of the security functions specified above:

- The scenarios for application of specified security functions/enablers
- The specification of relevant functions and their designs into modular components
- The software architecture block and sequence diagrams
- The relevant policies to realise the functionality of each security component
- The interfaces and interactions with other components
- The code and configuration of components
- The implementation and realisation of components for PoC experimentation purposes
- Identification of tests to be conducted for PoC
- Component-level PoC tests conducted in-house at each partner’s premises and the results obtained.

Future directions are also specified to further the work in each of the activities within the WP4 tasks and to provide the implemented security functions and enablers for integration and tests to WP6. Given the above aspects, we tried to build the case for “ease of use” and “ease of configuration” of security components for their installation and integration in WP6 scenarios.

Table of Contents

List of Figures

Acronyms

ABAC	Attribute Based Access Control
AC	Access Control
ACM	Access Control Manager
AP	Application Process
BPC	Boundary Protection Component
CA	Certificate Authority
CACEP	Common Application Connection Establishment Protocol
CBAC	Capability Based Access Control
CDAP	Common Distributed Application Protocol
CRC	Cyclic Redundancy Check
CTR	Counter
DAF	Distributed Application Facility
DAP	Distributed Application Process
DH	Diffie-Hellman
DIF	Distributed IPC Facility
DMS	Distributed Management System

DTCP	Data Transfer Control Protocol
DTLS	Datagram Transport Layer Security
DTP	Data Transfer Protocol
EFCP	Error Flow Control Protocol
FA	Flow Allocator
FLD	Flow Liveness Detection
FSDB	Flow State Database
FSO	Flow State Object
HMAC	Hash-based Message Authentication Code
IRATI	"Investigating RINA as an Alternative to TCP/IP" project
KFA	Kernel Flow Allocator
IPC	Inter Process Communication
IPCM	Inter Process Communication Manager
IPCP	Inter Process Communication Process
LB	Load Balancing
LBR	Load Balancer
LFA	Loop Free Alternates

MA	Management Agent
MAC	Message Authentication Code
MD5	Message Digest algorithm
MLS	Multi Level Security
OAEP	Optimal Asymmetric Encryption Padding
OSI	Open Systems Interconnection
PCI	Protocol-Control-Information
PDP	Policy Decision Point
PDU	Protocol Data Unit
PEP	Policy Enforcement Point
PFT	PDU Forwarding Table
PKI	Public Key Infrastructure
PoC	Proof of Concept
RBAC	Role-Based Access Control
RIB	Resource Information Base
RINA	Recursive InterNetwork Architecture
RINASim	RINA Simulator

RMT	Relaying and Multiplexing Task
RSA	Encryption algorithm
RTT	Round Trip Time
SDU	Service Data Unit
SerDes	Serialisation/Deserialisation
SHA	Secure Hash Algorithm
SP	Shortest Path
TLS	Transport Later Security
TTL	Time To Live
VM	Virtual Machine
WP	Work Package
XML	eXtensible Markup Language

1. Introduction

This deliverable will provide initial specifications, design, and implementations of innovative security functions and reliability enablers. It covers the functions and enablers described in D4.1 [D4.1] and the derived security mechanisms and functions developed within WP4 to enable more secure and reliable networks than those that we have today. These mechanisms and functions include the authentication, access control, encryption, and self-healing aspects to be utilised in RINA-based networks. The deliverable describes in each section the specification, design, the analysis, and Prof of Concept (PoC) implementations of these mechanisms and functions; addressing the security requirements of the scenarios analysed in D2.1 [D2.1]. At the end of each section, we draw the next steps for the specific function.

1.1. Specification and System Design

One of the major objectives of the PRISTINE project is to develop and evaluate the concepts, the architecture, functions and mechanisms for deploying and providing end-to-end security. WP2 deliverables described the overall PRISTINE reference architecture. Subsequently, deliverable D4.1 provided the overall PRISTINE functional security architecture and specifies each of the main security functions and the interactions among them. This deliverable presents the specification and system design by mapping and decomposing the functional security architecture and entities proposed in D4.1 into relevant components and system modules.

In this deliverable, we provide the following:

- Firstly, the software architecture in terms of block diagrams where possible for each component in terms of functions and internal/external interactions.
- Secondly, further decomposition of each of the components into modules of an implementation structure.
- Thirdly, the policies, code, files, and modules are organised in the development environment to build the component/modules considering modularity and their repetitive use and installation.

1.2. Implementation Tasks

Protecting the network and its resources (i.e., user data, management data and computing resources) from failures and attacks to disrupt the communication service are the main security objectives. Deliverable 4.1 provided the RINA security solution, the functions and the relevant enablers to achieve the above objectives. These functions and enablers included: Authentication, Access Control, Secure Channel and SDU Protection, Key Management functions, monitoring and countermeasures for reducing the security risks and combating the threats. D4.1 deliverable also looked at network resiliency and availability in RINA. In this deliverable, we provide the following in relation to the PoC implementation:

Six different authentication approaches were proposed in D4.1. Three of these are selected for design and implementation, namely *AuthNNone* (a simple policy with no authentication); *AuthNPassword* (a shared secret associated with the application name); and AuthNAsymmetricKey (a public key cryptography-based policy).

In D4.1, a DAF-based Capability Based Access Control model was explained and selected for design and implementation in PRISTINE. Further details of applying this approach to RINA and the implementation course are given in this deliverable.

In D4.1, multiple architectures to achieve Multi-Level Security (MLS) were presented and thoroughly discussed. Two common components are needed for these architectures namely “Crypto tunnelling device” and “Boundary Protection Component - BPC” were identified. We established that MLS-enabled network with only crypto tunnelling is possible, but limited to Multiple Single Levels. BPC allows applications on otherwise separate networks to communicate, subject to configured constraints. In this deliverable, we establish the implementation scenarios for two cases using the above components.

SDU protection is to protect the integrity and confidentiality of traffic when passed on to an underlying IPC Process. The required SDU protection algorithms/policies that are used and applied are described, implemented and reported in this deliverable.

Two architectural options (Centralised and Distributed) were suggested in D4.1 for assuming the role of Key Server as the security sensitive entity. We further refined these options and will discuss these choices in the next version of this deliverable.

We introduced a risk assessment methodology in D4.1 for combating threats and vulnerabilities in RINA. We identified a comprehensive set of threats to the RINA assets, their impacts, the threat scenarios, the likelihood occurrence of each scenario, security risks and the associated Security Controls to reduce the risks to an acceptable level. We identified that a number of threats can be reduced by performing monitoring actions. A range of techniques and a variety of applications can be used to monitor and collect information for detecting and assessing vulnerabilities and attacks. These techniques and monitoring tools, their relevance, and their applications to the identified threats will be reported in the next deliverable.

Maintaining the network resiliency in the case of failures and attacks and ensuring high-availability of the network for providing assumed services are set as the main objectives for RINA. In this deliverable, implementation scenarios for improving routing resiliency are explained. Routing software specification and implementation are also described. We also look at how Load Balancing can be achieved in RINA.

The implemented components and related protocols are subject to experimentations for different purposes. In addition to the engineering counterparts of the functional entities, a set of adaptors may also be required to implement and interface to the testbed and, furthermore, a set of tools, such as monitoring and analysis tools are required to assist the testing activities.

1.3. Proof-of-Concept Experimentations

Proof-of-Concept experimentations are the essential aspect of PRISTINE work to the end of fulfilling overall project objectives. In PRISTINE, experimentation activities are carried out in realistic and possibly in simulated network environments, as appropriate to the aspect of the PRISTINE work under test and the experimentation objectives.

Evidently, the type of the experimentation environment (testbed or simulation) affects the nature of the releases coming out from the WP activities. For WP4 prototype releases, PRISTINE security solution is developed to apply in generic engineering environment according to the selected implementation technologies. This type of release is set for experimenting in testbeds and use in WP6 use-cases.

1.3.1. Experimentation Categories

As for their objectives, experimentation activities can fall under the following a number of recognised categories:

- Functional verification and validation experiments - the former is aiming at assessing feasibility of implementation and proving the correct functionality and the latter is for meeting the set requirements (defined in WP2) and validity of specifications.
- Integration experiments - is aiming at verifying that the developed components/ sub-systems function properly when they are put together. This also allows us to validate the developed system against functional specifications and requirements.

- Performance assessment experiments - aiming at assessing the behaviour of the aspect under test in a variety of network operations and environment set-ups and conditions. Behaviour can be assessed in terms of scalability, stability, sensitivity and yielded benefits/incurred cost; as such, corresponding experiments or simulation studies could be carried out.

Obviously, experimentation objectives are restricted by the capabilities of the experimentation environment. As such, some performance assessment experiments can only be carried out in a simulated networking environment, and not in a limited testbed environment. And, functional verification experiments better be carried out in a realistic environment for exhibiting the correct functionality of the system under test from network operation perspectives.

From a WP4 perspective, given that implementation activities are experimentation driven, experimentation focus poses the requirement that, in addition to PRISTINE functional security aspects, appropriate tools may need to be used as required for fulfilling experimentation objectives.

In summary, WP4 produces prototype releases of components subjecting them to component-level functional verification/validation tests in the testbeds as well as providing appropriate interfaces facilitating integration to WP6 use-cases for further PoC experimentation.

1.3.2. Test Groups and Structure of Test Campaigns

We can divide the tests in three distinct groups:

- Component-Level Tests: these tests are conducted in-house at each partner's permises. The emphasis on these tests is set to perform functional validation and verification and performance assessment of individual components, algorithms, and processes. These tests are conducted in-house at WP4 for security components, normally with no interactions with other PRISTINE system components.
- Integration Tests for use cases: These tests are performed to validate and verify the integrated components coming from the technical WPs inter-work and function together (including middleware, interfaces, applications, etc.). These tests will be conducted in WP6, in a defined location, realising use-case scenarios.
- System Level Tests: The tests are conducted to prove the functionality and validating the correct behaviour of the entire network system collectively. These tests also determine whether the overall performance objectives of the proposed system is realised. These tests will also be conducted in WP6.

The PoC experimentation activities can use a common structure/template where possible, along the following lines:

- *Objectives:* Outlining the aspects under test (specified component, mechanism, algorithm, protocol) and the particular goals and benefits of experimentation.
- *Performance Metrics:* Specifying the metrics inherent to the particular functional aspect under test that quantify the experimentation objectives such as processing time, overhead, etc. are described. These metrics can be measured, through probes or through test tools.
- *Controlled Variables:* Specifying the configuration parameters of the aspect under test. The performance metrics will be calculated as a function of these configuration parameters.
- *Uncontrolled Variables:* Identifying the parameters of the external environment where the aspect under test is to operate affecting its behaviour and/or its performance. Such parameters are the topology, volume of traffic, etc.
- *Experimentation Environment:* Providing the platform and the set-up environment upon which the experimentation is to be carried out including the modules, the platform and required test tools, their capabilities and interactions.
- *Test Campaigns*: This is to specify the tests to be carried out in achieving the specified objectives. Each of the tests aims at verifying/assessing a particular aspect of the behaviour/performance of the functional aspect under test (quantified by appropriate metrics) in a variety of test cases (quantified by appropriate combinations of uncontrolled variables) as a function of its configuration parameters (quantified by appropriate controlled variables). Tests are aggregated in test suites according to the general category they fall in.

2. Authentication of IPC Processes

One of the first measures to implement for securing a distributed system is authentication. DIFs are securable containers, therefore in order to verify the identity of IPC Processes that want to join a DIF, proper authentication policies must be put in place. Such policies can range from no authentication (for trusted environments in which security is not a concern) to sophisticated policies that exploit cryptographic techniques for more hostile environments. Even within a single DIF, different regions of the DIF may use different authentication policies depending on the properties of the N-1 DIFs the IPC Processes are relaying on, as shown in the example of Figure 1, “Multi-provider DIF configuration”. The multi-provider DIF on top is floating over multiple N-1 DIFs: the *access DIF*, allowing customers to connect to the Provider 1’s IPC Process (IPCP) at the border router; or the *Provider 1 Regional DIF* connecting together all the IPCPs in the Provider 1’s border routers facing customers. Flows between IPCP A and IPCP B go over the N-1 DIF called *access DIF*, which is shared between the provider and its customers. Due to this shared nature, IPCPs A and B will probably use authentication policies that rely on strong cryptographic techniques, which also generate secure keys to encrypt the data exchanged over the *access DIF*. However, IPCP B and IPCP C use the *Provider 1 Regional DIF* to communicate. Since this DIF is in full control of the provider (joining it requires getting physical access to a provider facility), authentication may not be required at all or may be very simple (a shared password approach for example).

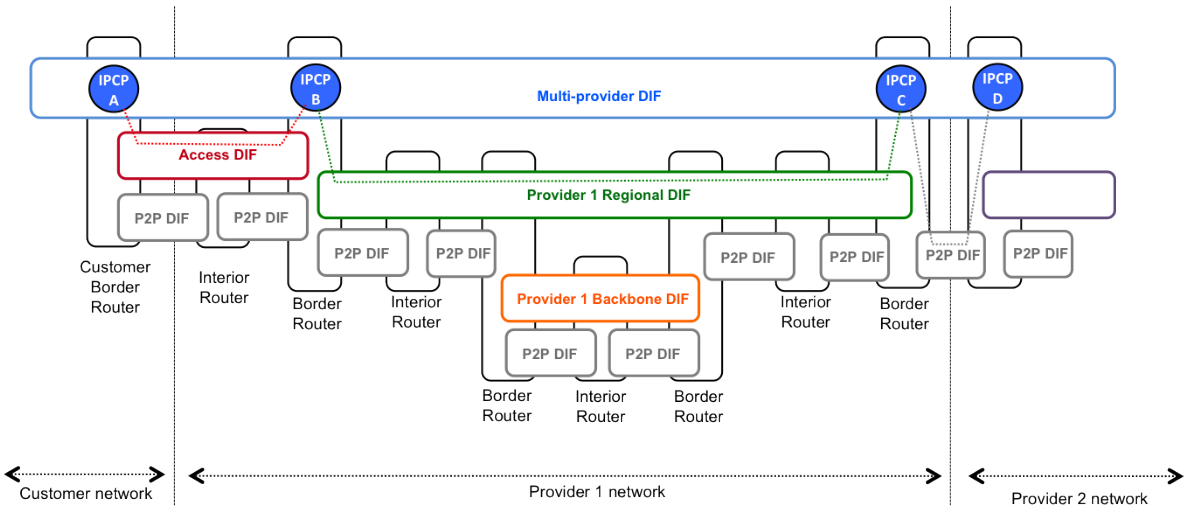

Figure 1. Multi-provider DIF configuration

Therefore, the authentication policies used by an IPCP may depend on the requirements of the DIF, the characteristics of the N-1 DIF or the type of system the IPC Process is executing on (host, interior router or border router). The goal of D4.2 with regards to authentication is to describe a few authentication policies that are representative of the full solution space; provide an initial specification of such policies; implement them at the IRATI RINA implementation leveraging PRISTINE's SDK; and validate its correct operation. D4.2 has focussed on the draft description of three of the authentication policies introduced in [D4.1], namely:

- **AuthNone**. The null case in which authentication is not required.
- **AuthNPassword**. The two IPC Processes authenticate by proving they know a previously shared password.
- **AuthNAssymetricKey (RSA)**. The two IPC Processes use cryptographic techniques and Public Key Infrastructure for authentication purposes. As a result of the authentication procedure, an encryption key is generated for the application connection and encryption is enabled.

2.1. Specification and Design of the Authentication Function

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 2, "Authentication between APs when establishing an application connection". 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).

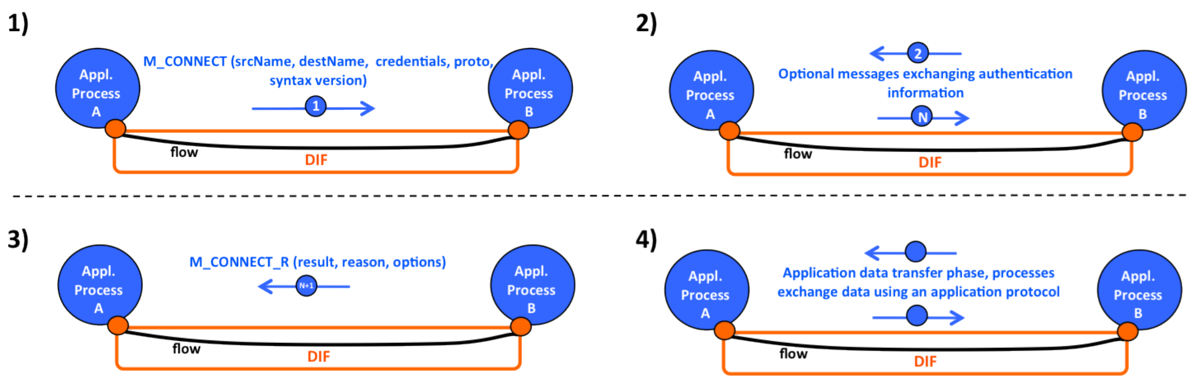
1)
Appl. Process A
M_CONNECT (srcName, destName, credentials, proto, syntax version)
1
Appl. Process B
flow
DIF
2)
Appl. Process A
2
Optional messages exchanging authentication information
N
Appl. Process B
flow
DIF
3)
Appl. Process A
M_CONNECT_R (result, reason, options)
N+1
Appl. Process B
flow
DIF
4)
Appl. Process A
Application data transfer phase, processes exchange data using an application protocol
Appl. Process B
flow
DIF

Figure 2. 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.

2.1.1. Specification of Three Authentication Policies

The three policies leverage the *'AuthPolicy'* field present in the CDAP M_CONNECT message. This field allows the party that initiates the application connection establishment to request a specific version of a particular authentication policy. The *'AuthPolicy'* field has three attributes:

- **Name**: a string that uniquely identifies the authentication policy name.
- **Versions**: an array of string specifying the versions of the policy supported by the party that requests the establishment of the application connection.
- **Options**: an optional opaque attribute that carries extra policy-specific information.

For the sake of brevity and clarity in the description of the specifications, we'll refer to "IPCP A" as the IPC Process that initiates the application connection request, and "IPCP B" as the IPC Process that is the target of the application connection request. Note that these specifications are not specific to a DIF and can be re-used by any type of DAF that considers these policies appropriate for its authentication requirements.

2.1.1.1. AuthNone Policy

Figure 3, "Workflow of AuthNone policy" illustrates the workflow of this authentication policy. IPCP A populates the *'AuthPolicy'* field with the following data:

- **Name**: PSOC_authentication-none.
- **Versions**: 1 (only supported version as of now).
- **Options**: empty.

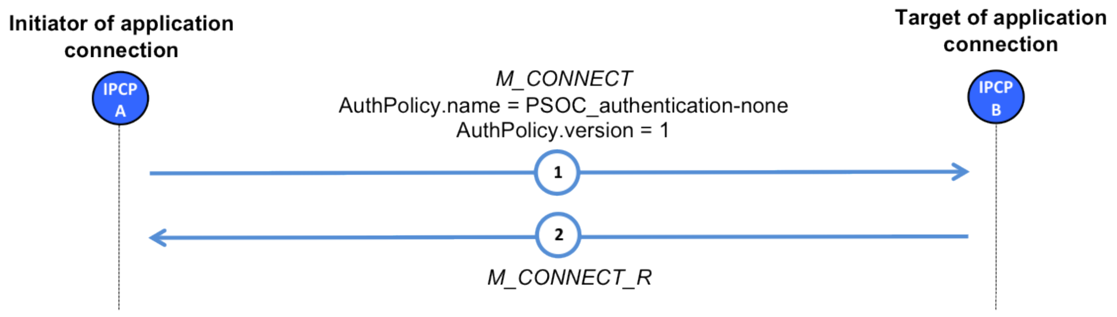
Initiator of application connection
IPCP A
Target of application connection
IPCP B
M_CONNECT
AuthPolicy.name = PSOC_authentication-none
AuthPolicy.version = 1
1
2
M_CONNECT_R

Figure 3. Workflow of AuthNone policy

Upon receiving the M_CONNECT message, IPCP B decides if the authentication policy is appropriate. If it is, it replies right away with a successful M_CONNECT_R message.

2.1.1.2. AuthNPassword Policy

Figure 4, "Workflow of AuthNPassword policy" 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.
- **Versions**: 1 (only supported version as of now).
- **Options**: empty.

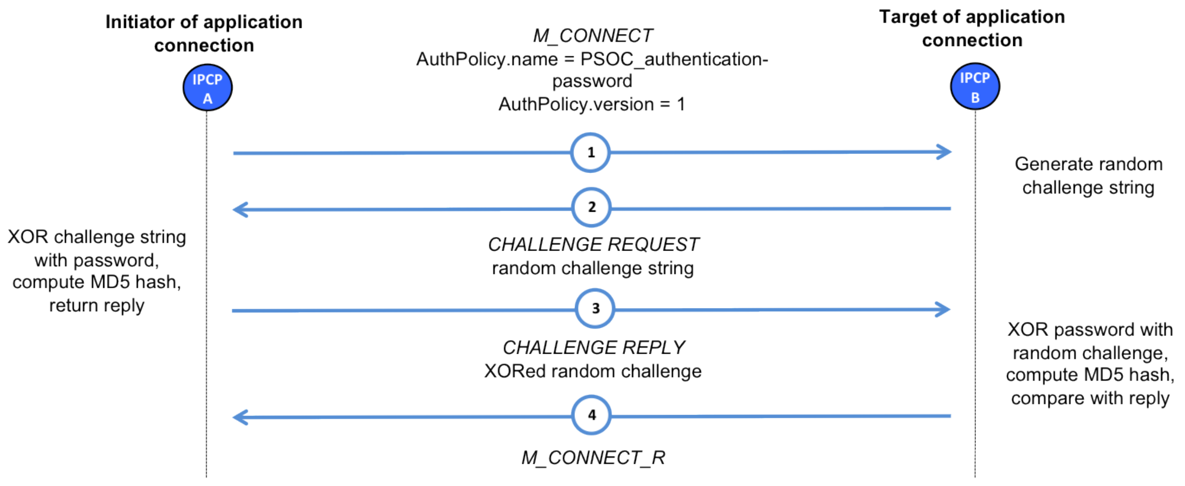
Initiator of application connection
IPCP A
Target of application connection
IPCP B
M_CONNECT
AuthPolicy.name = PSOC_authentication-password
AuthPolicy.version = 1
1
Generate random challenge string
2
CHALLENGE REQUEST
random challenge string
XOR challenge string with password, compute MD5 hash, return reply
3
CHALLENGE REPLY
XORed random challenge
XOR password with random challenge, compute MD5 hash, compare with reply
4
M_CONNECT_R

Figure 4. 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.

2.1.1.3. AuthNAssymetricKey (RSA) Policy

Figure 5, “Workflow of AuthNAssymetricKey (RSA) policy” illustrates the workflow of this authentication policy. It is inspired by the SSH2 Transport [RFC4253] and Authentication [RFC4252] protocols. The policy has two differentiated phases: in the first phase both parties securely negotiate a shared secret using the Diffie-Hellman (DH) key exchange method [DH]. This shared secret is then used to generate an encryption key to encrypt all the communication between both parties. DH is used in ephemeral mode (new shared secret generated for each application connection), with the advantage of generating shared secrets on the fly in a secure way; at the cost of one extra round trip time (RTT). An alternative to this approach would be to use a pre-shared secret, thus avoiding the RTT consumed by the DH key exchange but complicating the shared secret management and distribution (must be distributed in a secure way, should be updated after a certain period of time, etc.)

During the second phase both parties use PKI, specifically RSA, to authenticate its peer. The policy assumes the same RSA key pair for both IPCPs (A and B), but could also be modified to support different RSA key pairs for each party. During the authentication phase both IPCPs authenticate each other.

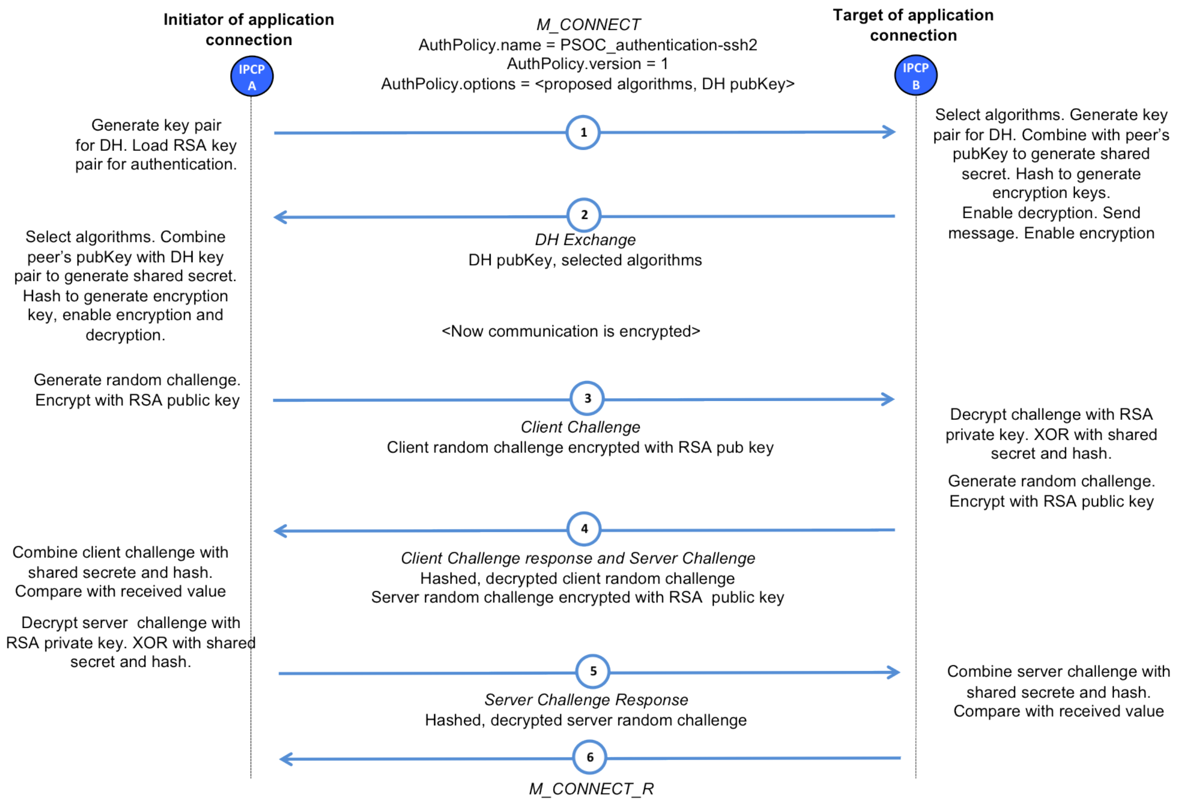
Initiator of application connection
IPCP A
M_CONNECT
AuthPolicy.name = PSOC_authentication-ssh2
AuthPolicy.version = 1
AuthPolicy.options = <proposed algorithms, DH pubKey>
Target of application connection
IPCP B
Generate key pair for DH. Load RSA key pair for authentication.
1
Select algorithms. Generate key pair for DH. Combine with peer's pubKey to generate shared secret. Hash to generate encryption keys. Enable decryption. Send message. Enable encryption
2
DH Exchange
DH pubKey, selected algorithms
Select algorithms. Combine peer's pubKey with DH key pair to generate shared secret. Hash to generate encryption key, enable encryption and decryption.
<Now communication is encrypted>
Generate random challenge. Encrypt with RSA public key
3
Client Challenge
Client random challenge encrypted with RSA pub key
Decrypt challenge with RSA private key. XOR with shared secret and hash.
Generate random challenge. Encrypt with RSA public key
4
Client Challenge response and Server Challenge
Hashed, decrypted client random challenge
Server random challenge encrypted with RSA public key
Combine client challenge with shared secrete and hash. Compare with received value
Decrypt server challenge with RSA private key. XOR with shared secret and hash.
5
Server Challenge Response
Hashed, decrypted server random challenge
Combine server challenge with shared secrete and hash. Compare with received value
6
M_CONNECT_R

Figure 5. Workflow of AuthNAssymetricKey (RSA) policy

IPCP A generates a DH key pair of length 256 bytes using pre-defined values of the parameters 'p' and 'g' required by the DH scheme ('p' and 'g' are not secret and typically take tens of seconds to be generated, therefore they must be static for a practical solution). Then IPCP A populates the *'AuthPolicy'* field with the following data:

- **Name**: PSOC_authentication-ssh2.
- **Versions**: 1 (only supported version as of now).
- **Options**: <list of supported Key exchange algorithms (only DH), list of supported encryption algorithms (AES128 and AES256), list of supported MAC algorithms (MD5 and SHA1), generated DH public key>

Upon receiving the M_CONNECT message, IPCP B decides if the authentication policy is appropriate. If it is, it checks the algorithms proposed by the client, and selects one of them for each category. If there are multiple options, IPCP B selects the first one that it supports (IPCP A must send the list of algorithms sorted by preference). After that, IPCP B generates a DH key pair, and combines it with IPCP A's DH public key to generate a shared secret. Then the secret is hashed to generate the encryption key (with the MD5 algorithm [RFC1321] if the encryption key is 16 bytes long, or with the SHA-256 algorithm [sha2] if the encryption key is 32 bytes long). Then IPCP B enables decryption, sends the following message to IPCP A and enables encryption (in this sequence, to avoid race conditions).

- **Opcode**: M_WRITE.
- **Object class**: Ephemeral Diffie-Hellman exchange.
- **Object value**: <Key exchange algorithm (only DH), encryption algorithm, MAC algorithms, generated DH public key>

When IPCP A receives the message, it uses IPCP B's DH public key to generate the shared secret, and after that the encryption key using the same approach as described before. Then IPCP A enables both encryption and decryption. From now on, all communication between A and B over the N-1 flow will be encrypted. After encryption is setup, IPCP A generates a random byte array of the same length of the DH shared secret (256 bytes). It then encrypts this number with the RSA public key, using Optimal Asymmetric Encryption Padding (OAEP), and sends it to IPCP B using the following message.

- **Opcode**: M_WRITE.
- **Object class**: Client challenge.
- **Object value**: <Client random challenge encrypted with RSA key>

IPCP B receives the message, decrypts the array of bytes with the RSA private key and XORs the result with the shared secret generated via the DH exchange. It then computes a 16 bytes hash of the result using the MD5 algorithm. IPCP B also generates a random byte array of 256 bytes and encrypts it with the RSA public key. Both values are sent back to the client using the following message.

- **Opcode**: M_WRITE.
- **Object class**: Client challenge reply and server challenge.
- **Object value**: <Client challenge combined with shared secret and hashed, Server random challenge encrypted with RSA key>.

When IPCP A receives the message, it XORs the client challenge that it had previously generated with the shared secret and computes the MD5 hash of the result. This value is compared with the value received form IPCP B. If they match IPCP B has proved it has the RSA private key and is therefore authenticated, if not IPCP A sends an M_RELEASE messate to IPCP B. Assuming a successful authentication, now IPCP A tries to decrypt the random challenge sent by IPCP B using the private key, XORs the result with the shared secret and computes the MD5 hash of the result. The value is delivered to IPCP B using the following message.

- **Opcode**: M_WRITE.
- **Object class**: Server challenge reply.
- **Object value**: <Server challenge combined with shared secret and hashed>.

Upon receiving the message IPCP B XORs the server challenge that it had previously generated with the shared secret and computes the MD5 hash of the result. This value is compared with the value received form IPCP A. If they match IPCP A has proved it has the RSA private key and is therefore authenticated. If authentication is successful IPCP B 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.

2.1.2. Interfaces and Interactions with Other Components

Figure 6, "Interaction between different application components" shows, at an abstract level, the main application components that are related to the authentication procedures and the main interactions amongst them. The image is not proposing any implementation design, it is just purely for a better understanding of authentication in the context of the DIF/DAF theory (multiple implementation strategies are possible).

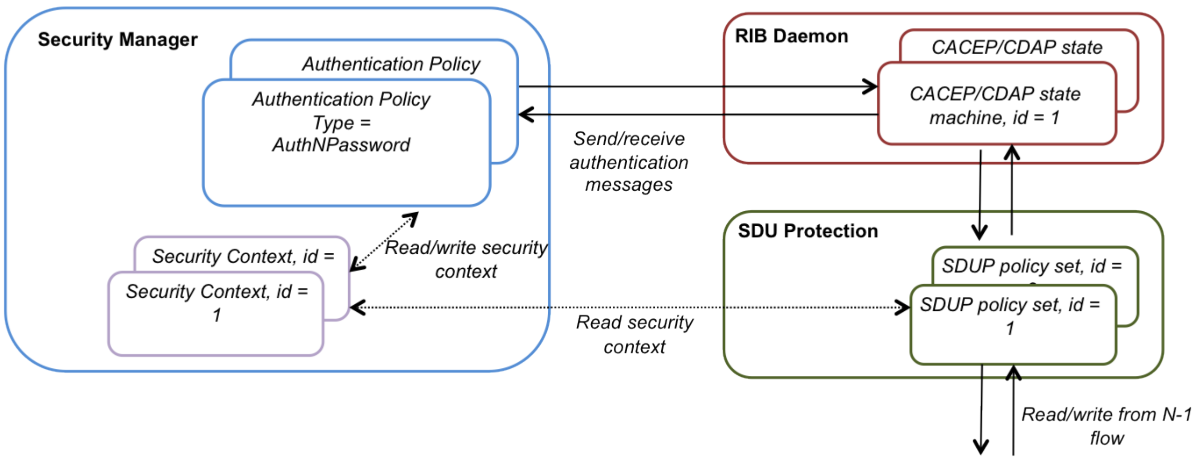

Figure 6. Interaction between different application components

There are three main components that are relevant to an application's authentication: the Security Manager, the RIB Daemon and the SDU Protection module.

- **SDU Protection module**: Protects/unprotects the data coming in/out an N-1 flow. Must be configured with the right policies and policy parameters (encryption algorithm, encryption key, etc.). The SDU Protection module configuration can be different for each different N-1 flow, and is owned by the Security Manager. The SDU Protection module can query a security profile to learn the operations that must be applied to incoming and outgoing SDUs.

- **RIB Daemon**. Receives incoming SDUs from SDU protection, which are CDAP messages targeting one or more RIB objects. The RIB Daemon is also the responsible for establishing an application connection to a remote application (encapsulating the CDAP and CACEP state machines). Before starting the application connection request, the RIB Daemon must query the Security Manager to obtain support of the relevant authentication policy module associated to the application connection. Any authentication-related messages received between M_CONNECT and M_CONNECT_R will be delivered to the authentication policy for its processing.
- **Security Manager**. Hosts all the authentication policy instances supported by the application, as well as the current security contexts (for each allocated N-1 flow). The authentication policy is in charge of initializing and populating the security profile associated with a particular N-1 flow with the relevant data (algorithms, key material, protection policies, etc). The authentication policy interacts with the RIB Daemon to send/receive authentication-related messages.

2.2. Implementation of the Authentication Function for PoC

The three authentication policies previously specified in this document have been implemented in *librina*, so that they can be used by an IPC Process but also by other application processes that follow the DAF model. The high-level design of the implementation roughly follows the model described in the previous section, taking into account the particularities of the IRATI RINA implementation: the IPC Process's SDU Protection module is located at the kernel, while the RIB Daemon and the Security Manager are at user-space. This makes the implementation design a bit more complex than what is explained in the high level model, since the security context state must be split between user-space and the kernel, while configuration of the SDU Protection module requires asynchronous messaging (via Netlink sockets).

2.2.1. Authentication-related SDK

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 AuthPolicy get_auth_policy(int session_id,
        const AuthSDUProtectionProfile& profile) = 0;

 /// 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 AuthPolicy& auth_policy,
         const AuthSDUProtectionProfile& profile,
         int session_id) = 0;

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

 //Called when encryption has been enabled on a certain port, if the call
 //to the Security Manager's "enable encryption" was asynchronous
 virtual AuthStatus encryption_enabled(int port_id) = 0;

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

The policy has to implement the following main operations:

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

- **encryption_enabled**. Callback informing about the result of an "enable encryption" 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).

2.2.2. Configuration of the Security Manager

The work reported in D4.2 has unified the configuration of the Security Manager and updated the format of the configuration file. The following code snippet shows an example configuration.

```
{
    "securityManager" : {
     "newFlowAccessControlPolicy" : {
          "name" : "default",
          "version" : "0"
        },
        "difMemberAccessControlPolicy" : {
          "name" : "default",
          "version" : "0"
        },
        "authSDUProtProfiles" : {
           "default" : {
              "authPolicy" : {
               "name" : "PSOC_authentication-sshrsa",
           "version" : "1",
           "parameters" : [ {
              "name" : "keyExchangeAlg",
              "value" : "EDH"
           }, {
                   "name" : "keystore",
                   "value" : "/usr/local/irati/etc/private_key.pem"
                }, {
                   "name" : "keystorePass",
                   "value" : "test"
                } ]

              },
              "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"
                }
           },
           "specific" : [ {
               "underlyingDIF" : "100",
               "authPolicy" : {
                  "name" : "PSOC_authentication-none",
              "version" : "1"
                }
           }, {
               "underlyingDIF" : "110",
               "authPolicy" : {
                  "name" : "PSOC_authentication-password",
              "version" : "1",
              "parameters" : [ {
                 "name" : "password",
                 "value" : "kf05j.a1234.af0k"
              } ]
                },
                "TTLPolicy" : {
                   "name" : "default",
                   "version" : "1",
                   "parameters" : [ {
                      "name" : "initialValue",
                      "value" : "50"
                   } ]
                },
                "ErrorCheckPolicy" : {
                   "name" : "CRC32",
                   "version" : "1"
                }
           } ]
        }
    }
 }
```

The first two fields are dedicated to the configuration of the *new member access control policy* (executed after successful authentication of a remote IPCP) and the *new flow access control policy* (executed when there is an incoming flow allocation request for an application registered in the IPCP). After that there is the configuration of the policies that can vary depending on the N-1 DIF supporting this IPCP. These policies are: authentication, encryption, error check and TTL. The Security Manager configuration provides a default and specific sets of these policies (the default set is used whenever no N-1 DIF specific policy is specified).

2.2.3. AuthNone Policy

The implementation of the AuthNone policy is trivial. The **get_auth_policy** operation returns an *AuthPolicy* object populated with the information described in the policy sepecification. The **initiate_authentication** policy just checks for the correct policy names and version, and returns *SUCCESS*. The **process_incoming_message** and the **encryption_enabled** operations are not used and therefore just return *FAILURE* (they should not be called). The snippet below shows an example of the AuthNone policy configuration.

```
{
...
              "authPolicy" : {
               "name" : "PSOC_authentication-none",
           "version" : "1"
              },
...
```

2.2.4. AuthNPassword Policy

The **get_auth_policy** operation returns an *AuthPolicy* object populated with the information described in the policy sepecification. The **initiate_authentication** policy checks for the correct policy names and version, generates a random string of the same length as the password, asks the RIB Daemon to send a CDAP message to the remote IPCP and returns *IN PROGRESS*. The **process_incoming_message** operation processes the two different messages involved in this policy: the challenge message and the challenge request message, as described by the policy specification.

The **encryption_enabled** operation is not used and therefore just returns *FAILURE* (it should not be called). The snippet below shows an example of the AuthPassword policy configuration.

```
{
...
               "authPolicy" : {
                  "name" : "PSOC_authentication-password",
              "version" : "1",
              "parameters" : [ {
                 "name" : "password",
                 "value" : "kf05j.a1234.af0k"
              } ]
                },
...
```

2.2.5. AuthNAssymetricKey (RSA) Policy

Since a number of cryptographic operations have to be performed by this authentication policy, it needs to rely on a well-accepted implementation of these functions. The openSSL libcrypto library [openssl] has been chosen as a provider of cryptographic functions for the user-space IRATI daemons, due to its widespread use and completeness of the implementation. In particular, this policy uses the following facilities provided by libcrypto: Diffie-Hellman key and shared secret generation, MD5 and SHA-256 hash functions, loading RSA keys from PEM files, RSA public key encryption and private key decryption.

The **get_auth_policy** operation returns an *AuthPolicy* object populated with the information described in the policy specification (including the DH public key). The **initiate_authentication** policy checks for the correct policy names and version, selects the algorithms to be used for encryption, generates the DH key-set and the shared secret (with associated encryption key). Once this is done it asks the Security Manager to enable decryption on the N-1 port (which is an asynchronous operation).

The **enable_encryption** operation is invoked when the kernel has replied to an enable encryption request. It considers three cases: IPCP B had asked to enable decryption, IPCP B had asked to enable encryption or IPCP A had asked to enable both encryption and decryption. In the first case the policy sends a "DH exchange message" to IPCP A, with IPCP B’s DH public key. In the second case a condition variable is updated (notifying that encryption is completely setup). In the last case IPCP A generates the challenge byte array, encrypts it with the public RSA key and sends it to IPCP B.

The **process_incoming_message** operation processes the four different messages involved in this policy: the *DH exchange message*, the *client challenge message*, the *client challenge reply message with server challenge* and the *server challenge reply message*.

- **DH exchange message**. IPCP A computes the shared secret and encryption key, requesting both encryption and decryption to be enabled for the related N-1 port in the kernel. Once the answer is obtained IPCP A proceeds as explained in the last paragraph.
- **Client challenge message**. IPCP B decrypts the challenge with the private RSA key, XORs it with the shared secret and computes the MD5 hash. It also generates a random byte array (the server challenge) and sends both values back to IPCP A.
- **Client challenge reply and server challenge**. IPCP A XORs the client challenge that was sent to IPCP B with the shared secret, computes the MD5 hash and compares it with the client challenge reply. If they are equal IPCP B has been successfully authenticated, if not an M_RELEASE is sent to IPCP B and the operation returns *FAILURE*. In the case when both values were equal, IPCP A decrypts the server challenge with the private RSA key, XORs it wit the shared secret, computes the MD5 hash and sends it to IPCP B.
- **Server challenge reply**. The received challenge reply is verified following the usual procedure described in the former paragraph, resulting in a successful or failed authentication of IPCP A (the operation returns *SUCCESS* or *FAILED* accordingly).

The snippet below shows an example of the AuthNAssymetricKey (RSA) policy configuration, as well as of the associated encryption policy that must be activated for the N-1 port. The authentication policy needs to be populated with information on the key exchange algorithm (right now only Diffie Hellman on Ephemeral mode is supported), the location of the file with the RSA key, and the password to be able to read the RSA key from the file, since it is encrypted (NOTE: this feature is still missing in the PoC as of D4.2 writing, but will be implemented in short; until then keys are stored in the clear).

```
{
...
              "authPolicy" : {
               "name" : "PSOC_authentication-sshrsa",
           "version" : "1",
           "parameters" : [ {
              "name" : "keyExchangeAlg",
              "value" : "EDH"
           }, {
                   "name" : "keystore",
                   "value" : "/usr/local/irati/etc/private_key.pem"
                }, {
                   "name" : "keystorePass",
                   "value" : "test"
                } ]

              },
              "encryptPolicy" : {
                 "name" : "default",
                 "version" : "1",
                 "parameters" : [ {
              "name" : "encryptAlg",
              "value" : "AES128"
           }, {
              "name" : "macAlg",
              "value" : "SHA1"
           }, {
              "name" : "compressAlg",
              "value" : "default"
                } ]
              },
...
```

2.3. Component-Level PoC Tests for Authentication

The experimental scenario used to verify the correct operation of the *AuthNPassword* and the *AuthNAssymetricKey(RSA)* authentication policies is shown in Figure 7, "Authentication policies verification scenario". A normal DIF consisting of three IPCPs operates over two shim DIFs over Ethernet. IPCP *test3.IRATI* is configured to use the *AuthNPassword* authentication policy by default, with an Error Check (CRC) and TTL policies but without an encryption policy. IPCP *test2.IRATI* is configured to use the *AuthNAssymetricKey(RSA)* authentication policy by default, with Encryption, Error Check and TTL policies. However, it is also instructed to use the *AuthNPassword* authentication policy and no encryption for N-1 flows over the N-1 DIF called "100". IPCP *test3.IRATI* is configured to use always the *AuthNAssymetricKey(RSA)* authentication policy with Encryption, Error Check and TTL policies.

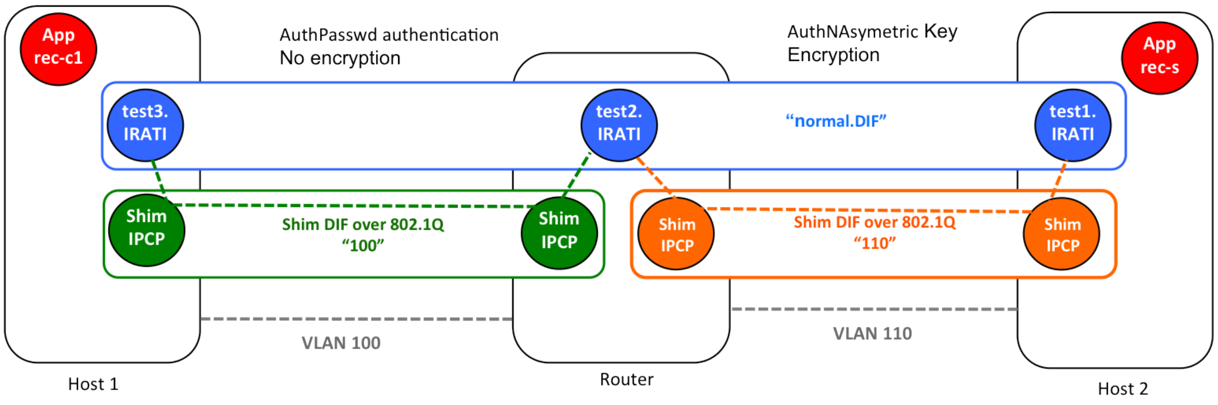

Figure 7. Authentication policies verification scenario

2.3.1. AuthNPassword Policy

The following traces are the output of capturing the Ethernet packets at the *eth1.100* interface of the system *Host 1* with the Linux utility *tcpdump*. ARP request and response correspond to the ARP request and reply issued by the shim DIF when the IPC Process *test3.IRATI* requests a flow allocation to the IPC Process *test2.IRATI*.

M_CONNECT message reflects *test3.IRATI* sending an M_CONNECT message to *test2.IRATI*, requesting a new connection to be opened using the 'PSOC_authentication_password' authentication policy with version '1'.

IPCP *test2.IRATI* replies with a *challenge request* message, providing the random string that *test3.IRATI* XORs with the password and sends back to IPCP *test2.IRATI* in a *challenge reply* message, as depicted by Challenge request and response messages.

Authentication is successful and IPCP *test2.IRATI* replies with an M_CONNECT_R message, as shown in M_CONNECT_R message. Then the enrollment procedure continues with more message exchanges between both IPCPs.

2.3.2. AuthNAssymetricKey (RSA) Policy

The following traces are the output of capturing the Ethernet packets at the *eth1.110* interface of the system *Host 2* with the Linux utility *tcpdump*. ARP request and response correspond to the ARP request and reply issued by the shim DIF when the IPC Process *test1.IRATI* requests a flow allocation to the IPC Process *test2.IRATI*.

M_CONNECT message reflects *test1.IRATI* sending an M_CONNECT message to *test2.IRATI*, requesting a new connection to be opened using the 'PSOC_authentication-ssh2' authentication policy with version '1'. The DH public key is also provided as part of the *options* field in the *AuthPolicy* field options.

IPCP *test2.IRATI* replies with the *Ephemeral Diffie-Hellman exchange* message, providing its DH public key to *test1.IRATI*. From now on, all messages are encrypted, as shown by the trace of the next packet in EDH exchange and encrypted client challenge message.

Since the communication is encrypted, showing the log of *tcpdump* is not very illustrative. IPCP *test1.IRATI* log shows the log of IPCP *test1.IRATI* (the one that initiated the application connection). The sequence of messages shows how *test1.IRATI* i) receives the *Ephemeral DH exchange* message form *test2.IRATI*; ii) generates the encryption key; iii) enables encryption and decryption; iv) sends the *Client challenge* message; v) receives the *Client challenge reply and Server challenge* message; vi) sends the *Server challenge reply* message and vii) receives an M_CONNECT_R message indicating that the application connection has been successfully established.

2.4. Next Steps for Authentication Activity

The authentication policies developed within WP4 will be used in the first iteration of experimental activities that are reported in [D6.1]. Feedback from these experiments will be incorporated into WP4 for further refinement. In addition to this, the research and development activities related to authentication during the second iteration of PRISTINE will tackle two main topics:

- The specification and development of an authentication policy inspired by the TLS Handshake protocol [RFC5246], which uses certificates to authenticate both parties. This authentication policy will be associated with an encryption policy equivalent to the TLS record protocol [RFC5246].
- The investigation of authentication in the context of a DIF, after the IPC Process has successfully joined the DIF.
    - Once the IPCP has authenticated with a DIF member, what should it do if it wants to create application connections with other DIF members in order to exchange layer management information? Should it use the same authentication policy used to join the DIF or can this requirement be relaxed?
    - IPCPs can request the allocation of layer management flows to peer IPCPs (dedicated to the exchange of layer management information via CDAP), and

also data transfer flows, which are dedicated to carry user traffic over EFCP. Therefore no application connection is setup over data transfer flows but, should there be some form of authentication anyway over those flows? Otherwise, how can the IPCP that is a target of a data transfer flow be sure about the identity of the requestor of the flow?

3. Capability-based Access Control

Capability based Access Control (CBAC) is the approach to access control adopted for the PRISTINE project, as decribed in D4.1. CBAC is defined to simplify administration of permissions for a large number of users. It could be implemented as either the classical Role Based Access Control (RBAC) or in the advanced Attribute Based Access Control (ABAC). The capability is computed based on the role, in case of RBAC, or attributes of the user, in case of ABAC.

RBAC models categorize users based on similar needs and group them into roles. Permissions are assigned to roles rather than to individual users. The objective is to reduce the number of assignments. The more users and permissions a single role captures, the greater the administrative efficiency gains. Ideally, users should be assigned permissions which at any point in time represent a true reflection of current business rules, risk-mitigating precautions and context-related security measures.

The ABAC approach defines a capability or authorization token as one of the attributes of the entity that requires access to a certain resource in the system. Whereas RBAC provides coarse-grained, predefined and static access control configurations, ABAC offers fine-grained rules which are evaluated dynamically in real-time.

In the scope of this work, we study the application of ABAC to RINA. ABAC is based on token generation that designates an object and grants the subject (i.e. the holder of the token) authority to perform actions on that object. It defines the name for identifying the object and the set of access rights for that object. The token could be seen as a ticket, if a subject possesses this ticket it has the proof of the holder's rights to access the object.

As depicted in Figure 8, "Attribute Based Access Control System Architecture", the ABAC system generates a token which will then be used, along with environment and resources attributes, as input to the AC policy to decide whether to permit or deny access.

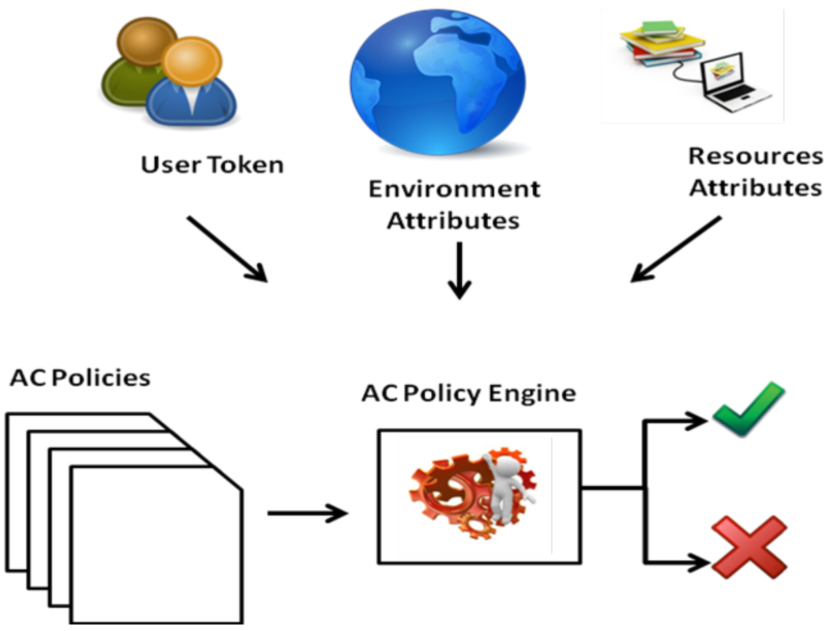

Figure 8. Attribute Based Access Control System Architecture

3.1. Access Control Scenarios

Access Control in PRISTINE is a crucial step that must be performed in different scenarios where requestors (subjects) would like to access to resources (objects). These scenarios are as follows:

- When an AP needs to access another AP's resources in the same DAF. In this case the peer AP should execute the AC function to permit or deny the requesting AP to access the requested objects.
- When an IPC Process requests to join a DIF, a check on the authorization rights of the requesting IPCP is needed.
- When an IPCP initiates the execution of remote operations on the objects of a peer's IPCP RIB.

The scenarios stated above may be processed in different ways and several AC policies could be applied for each case. In the scope of this deliverable, we will consider the first scenario. Note that most of the described procedures can be adapted to other scenarios. The remaining two scenarios will be specified and reported in the next WP4 deliverable (D4.3). In the following section, we will provide the specification and the design of the access control system for the first scenario.

3.2. Specification and Design of CBAC's at DAF-Level

We assume that any Distributed AP (DAP) acts as the subject that is is required to be authorized to proceed with some actions on the resources (objects) of other DAPs.

Objects concern the data and contents of the RIB within the DAP. Basically, the access control system will provide the corresponding capabilities to allow the requesting DAP to get access to the required resources. Figure 9, "AC System Architecture Block Diagram" shows the CBAC functional blocks and the interactions with RINA components. These blocks are explained below.

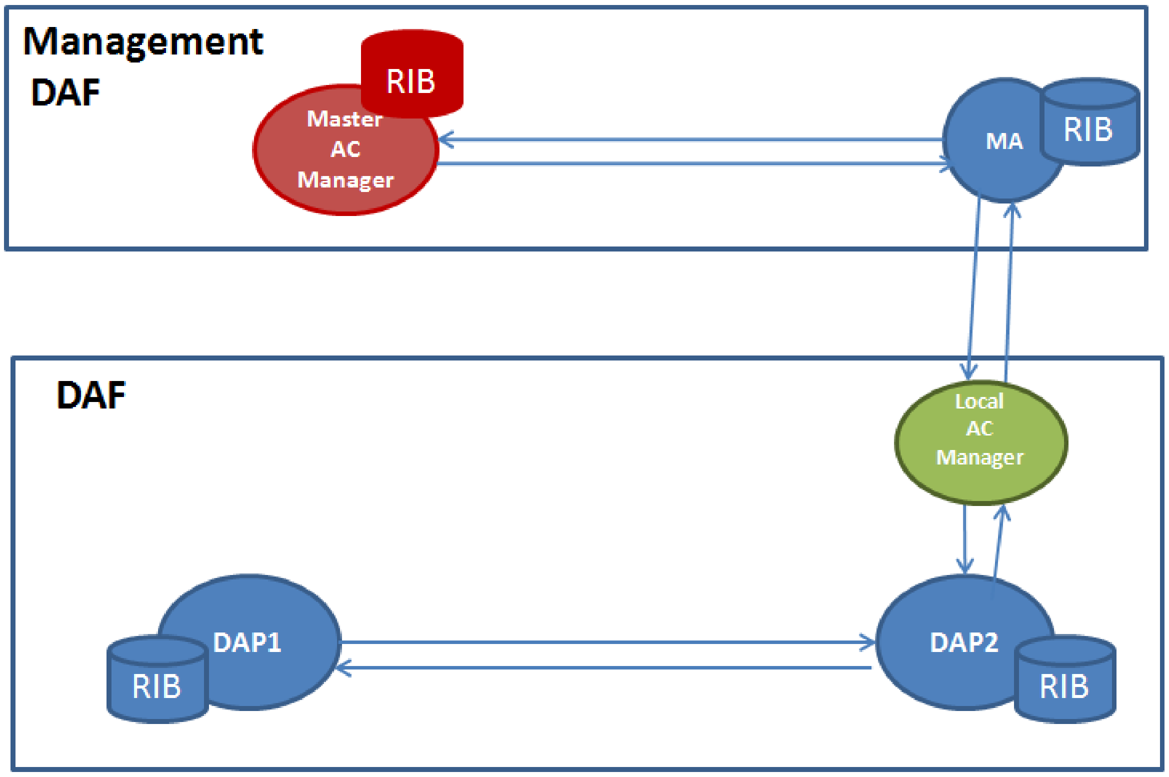

Figure 9. AC System Architecture Block Diagram

The originating DAP or the requestor: The service or application process requesting the RIB resources of the peer service. In our example DAP1

The receiver DAP: The service or application process having the requested RIB objects, e.g. a printing service. In our example DAP2.

The Management Agent (MA): The Management Agent is implemented as an Application Process (AP). Basically, here, it is responsible for providing the system with the needed access control data if it is not available in the AC Local Manager.

3.2.1. Access Control Mangers' Functions

3.2.1.1. The Master AC Manager

The AC master is the block responsible for storing the needed access control data including the authorization profiles of the different DAPs and the AC policies or rules that will be used then in the access control procedure. This entity operates in the same domain as the Distributed Management System (DMS)and could function in centralized or descentralized/replicated manner. It can be accessed via the DMS.

The information that must be stored in the AC Master block is the Authorisation Profiles.

3.2.1.2. The Local AC Manager

The AC Local Manager is the block implementing and enforcing the access control policy locally in the system that AP operates. The input to this block is the access control information that is requested from the AC Master via the MA (Management Agent). The output will be the AC decision and the eventual AC parameters that will be used in the AC procedure.

3.2.2. Authorisation Profiles

Profiles are stored in the RIB of the Master AC manager. They include:

- Profile name
- Profile type Generic_Profile for a given DAP, or Specific_Profile.
- Profile groups that the DAP belongs to
- Allowed objects description: Name, properties, accounting.

In the access control architecture we will define **four profiles that correspond to DAPs, RIBs, DAFs, and USERs**. Those profiles will be stored by the AC Manager Master. Each of them will be specified with a set of attributes. We define an attribute “group” that is assigned to different USERs or DAPs having similar access rights to different RIB objects.

In the case of DAPs, we define two groups and roles:

- S_GROUP: assigned to DAP servers that are able to execute certain services such as executing a program, providing certain services to other group called C_GROUP.
- C_GROUP: assigned to DAP Clients that will be asking for certain services from other DAPs. C_GROUP DAPs might be used by USERs requesting access to services offered by the DAF.
- We also define two roles: Management Agent and Application.

In the case of Users, we also define two groups and roles:

- A_Group for Users with high access rights such as Administrators.
- U_Group for users that are customers of the offered services in the DAF.
- We also define two roles USERACCESSONEHOUR, USERACCESSUNLIMITED.

3.2.2.1. Example Profiles

An example of defined profiles is given below. Consider a network NET1 where the RINA-enabled System1 has a DAF named DAF1 with two applications: DAP1 and DAP2. A Network Zone is defined as a network (NET1) under a single administrator. DAP1 application would like to access RIB information of DAP2. In this example DAP1 will play the role of Client to DAP2 which play the role of Server. Here, RoleD1 is a client role. We consider that these DAPs possess certificates. We consider User1 that uses DAP1 to access to services of DAF1. Some of the services are requesting access to the RIB2 of DAP2. We consider DAP3 and DAP4 as other application processes of the DAF1.

The authorisation profiles of DAF1, DAP1, User1, and RIB2 for this example are defined below:

```
<DAF profile starts>
{System “Name”: System1
DAF “Name”: DAF1
DAP « DAF » : DAF1
DAP “Network zone”: NET1
DAF “Certificate”:  CERTIFDAF1
DAF « creation date » : dd/mm/yyyy
DAF “end date” : dd/mm/yyyy
DAF “Ressources “: {RIB1, RIB2, …others}
DAF “Services”: { DAP2, DAP3, DAP4}
DAF “other profile information”: AddFunction
}
<DAF Profile ends>
```

```
<DAP profile starts>
{DAP “Name”: DAP1
DAP « DAF » : DAF1
DAP “group”:  C_Group
DAP “Role”:  Application
DAP “Password”: DPWD
DAP “Network zone”: NET1
DAP “Certificate”:  CERTIFDAP1
DAP « creation date » : dd/mm/yyyy
DAP “end date” : dd/mm/yyyy
DAP “Ressources “: {RIB_Public, RIB_Private, others}
DAP “other profile information”: AddFunction
}
<DAP Profile ends>
```

```
<RIB profile starts>
{RIB “Name”: RIB2
RIB « DAF » : DAF1
RIB “DAP”: DAP2
RIB “Password”: RPWD
RIB “Network zone”: NET1
RIB “Certificate”:  CERTIFRIB2
RIB « creation date » : dd/mm/yyyy
RIB “end date” : dd/mm/yyyy
RIB “other profile information”: AddFunction
}
<RIB Profile ends>
```

```
<USER profile starts>
{USER“Name”: User1
USER « DAF » : {DAF1, DAF2}
USER “DAP”: DAP1
USER “Role”: USERACCESSONEHOUR
USER “Password”: UPWD
USER “Certificate”:  CERTIFUser1
USER “other profile information”: AddFunction
}
<USER Profile ends>
```

3.2.3. Access Control Policies

Attribute evaluation enables effective **policy-based authorization**. In the architecture shown in Figure 9, “AC System Architecture Block Diagram”, we define two policies: **PERMIT** and **DENY** Policies. Consider the following three examples:

Example 1:

A **Policy** states that "all DAPs belonging to the DAF1 should have read access to RIB information located in a network zone NET1 made available to applications of the same DAF and running in the same network zone NET1 as the DAP.

An **access request evaluation** based on the following attributes and attribute values should therefore return **PERMIT**:

```
Subject's "DAF"="DAF1"
Subject's "Network Zone"="NET1"
Subject’s “Call_TokenFunction(Subject = DAP1, Object=RIB2)” =  “Authorise”
Action="read"
Resource "type"="RIB Information"
Resource "Network Zone"="NET1"
```

Note that “Call_TokenFunction(Object=RIB) “ in this example is the function that is called by the DAP2 which is applying the access control policy for requesting access to the RIB information by DAP1. If the result of this called Function is not authorized, then the applied policy will be “Not Permit”.

Example 2:

A **Policy** states that a user1 (defined in the profile earlier) in DAF1 of network zone NET1 requesting read access to the RIB2 resource of DAP2 in DAP1 but only for one hour will return **PERMIT**.

```
Subject's "DAF"="DAF1"
Subject’s “DAP”= “DAP1”
Subject's "Network Zone"="NET1"
Subject’s “Call_TokenFunction(Subject =User1, Object=RIB2)” =  “Authorise”
Action="read"
Resource "type"="RIB2"
Ressource “DAP”= “DAP2”
Ressource “DAF”= “DAF1”
Resource "Network Zone"="NET1"
```

Example 3:

A **Policy** states that a user1 (defined in the profile earlier) in DAF1 of network zone NET1 requesting read access to RIB2 resource of DAP2 for an unlimited time will return **DENY**.

```
Subject's "DAF"="DAF1"
Subject’s “DAP”= “DAP1”
Subject's "Network Zone"="NET1"
Subject’s “Call_TokenFunction(Subject =User1, Object=RIB2)” =  “Authorise”
Action="read"
Resource "type"="RIB2"
Ressource “DAP”= “DAP2”
Ressource “DAF”= “DAF1”
```

```
Resource "Network Zone"="NET1"
```

In this example Call_TokenFunction will return Not Authorized, as User1 role is defined with the access for only one hour and the requested access in this example is for unlimited access.

3.2.4. Interfaces and Interactions with Other Components

Figure 10, "DAP interactions with the Management DAF level" shows the AC procedure performed between two DAPs. It illustrates the interaction between RINA components. Each system has a MA (with its respective RIB and RIB daemon) in order to ensure the access to the Master AC Manager. The interaction between the MAs and Manager is based on CDAP messages.

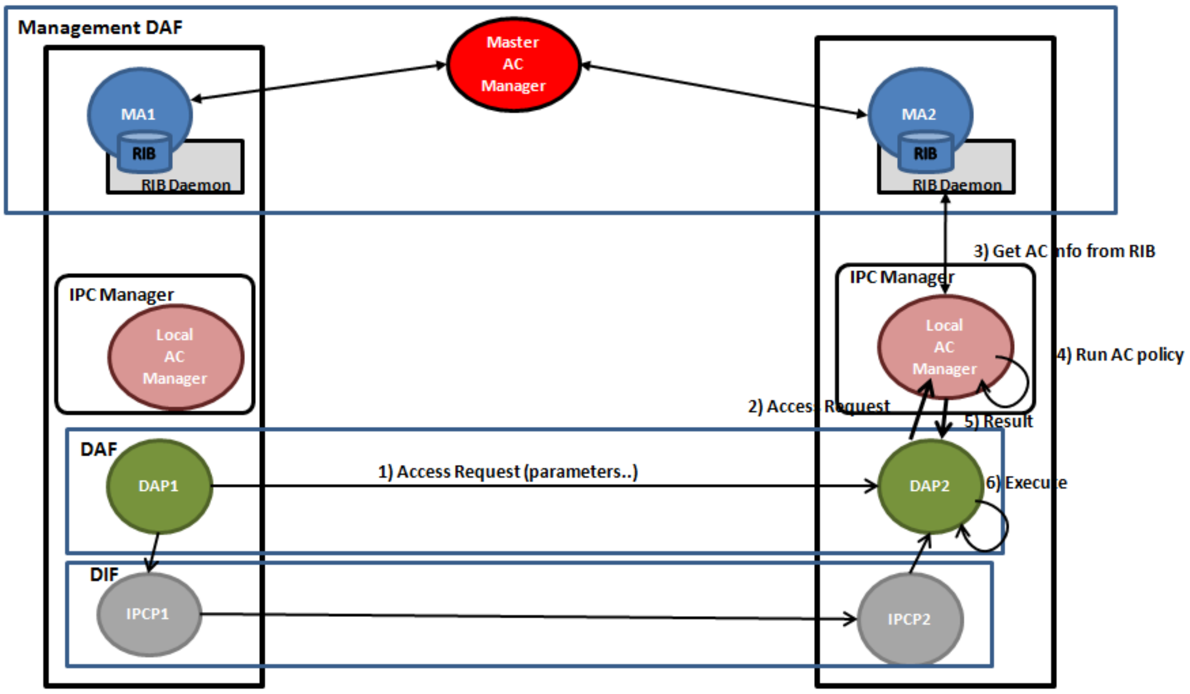

Figure 10. DAP interactions with the Management DAF level

3.2.4.1. Sequence Diagram and Interactions

Figure 11, "Sequence Diagram of the AC components' interaction" shows the sequence diagram of our scenario, illustrates these details, showing the interaction between the different components of the AC system.

Upon receiving the request from the originating DAP, the AC check procedure is launched on the destination DAP side:

1. The Local AC Manager requests the authorization profiles and AC policies or rules from the RIB Daemon of the Management Agent.

2. If the required information is not found in the local RIB, the MA RIB daemon should request it via CDAP from the AC Manager Master.
3. The Local AC Manager, located in the IPC Manager of the destination system, generates the Token, loads the policies and then inputs them to the Policy Decision Point (PDP). It should be noted that functions of PDP and PEP are described in the next section.
4. The PDP will output the AC decisions to the destination DAP.
5. The Policy Enforcement Point (PEP) enforces the decision and, if needed, performs an additional accounting check.

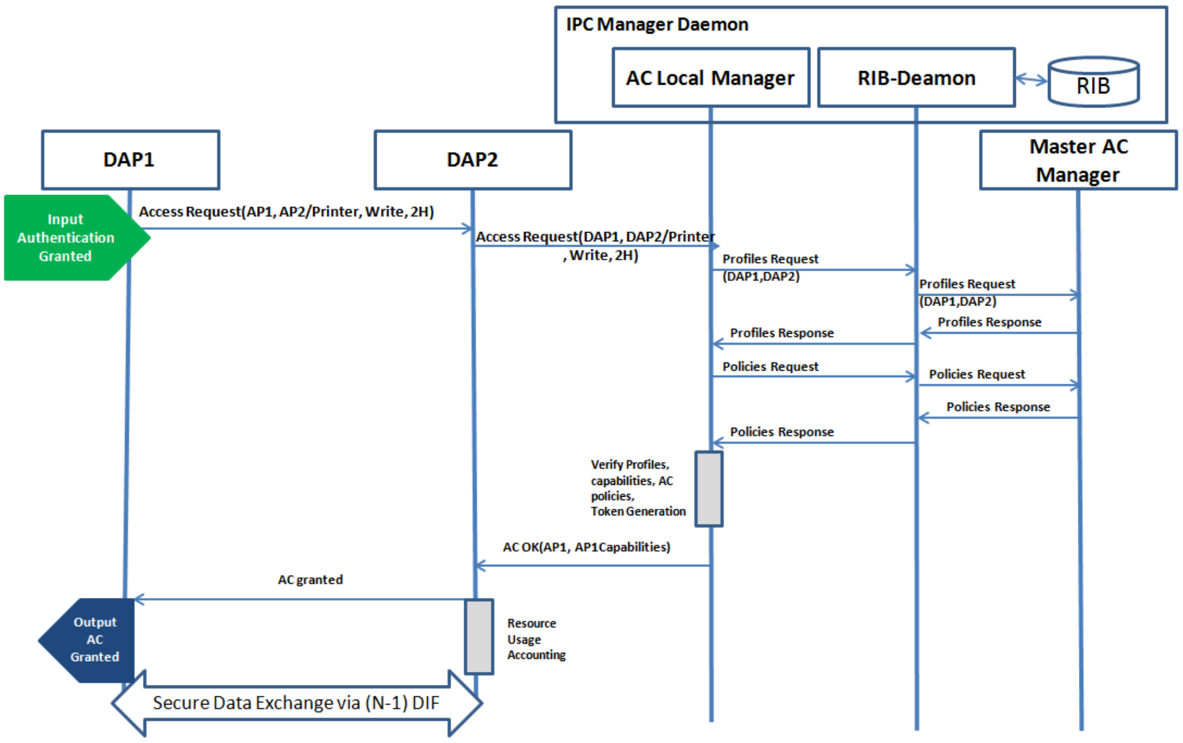
IPC Manager Daemon
AC Local Manager
RIB-Deamon
RIB
DAP1
DAP2
Master AC Manager
Input Authentication Granted
Access Request(AP1, AP2/Printer, Write, 2H)
Access Request(DAP1, DAP2/Printer, Write, 2H)
Profiles Request (DAP1,DAP2)
Profiles Request (DAP1,DAP2)
Profiles Response
Profiles Response
Policies Request
Policies Request
Policies Response
Policies Response
Verify Profiles, capabilities, AC policies, Token Generation
AC OK(AP1, AP1Capabilities)
AC granted
Output AC Granted
Resource Usage Accounting
Secure Data Exchange via (N-1) DIF

Figure 11. Sequence Diagram of the AC components' interaction

3.2.4.2. Inputs and outputs

Figure 12, "Inputs and outputs of the AC system" depicts the detailed specification of inputs/outputs of the different blocks of the architecture.

There are three actors:

- The Local AC Manager: this takes as inputs the Authorization profiles and the AC policies. This information should come from the RIB Daemon of the Management Agent. As the output, the Local AC Manager gives the AC Decision. There are three blocks inside the Local AC Manager :

  - PDP: the Policy Decision Point is the decision block where the system checks the profiles and the capabilities applying the AC policies in order to decide whether to permit or deny the access to the RIB resources. The inputs to this block are mainly the policies and the generated token. The output is the AC decision
  - Token Generation: This block is responsible of generating the token used afterwards in the PDP. The procedure of token generation will be detailed later.
  - AC Policies Loader: This module is in charge of loading the AC policies or rules from MA RIB Daemon. If the information is not found in the local RIB, the Master AC Manager is requested to provide it.
- The DAP processes AC via its PEP block which is responsible for ensuring the RIB Daemon respects the policy rules and changes its behaviour in accordance to the policies when needed. The input to this block is the AC decision.
- The AC Master Manager responds to the requests. The RIB daemon of the MA sends CDAP messages to the AC Master Manager.

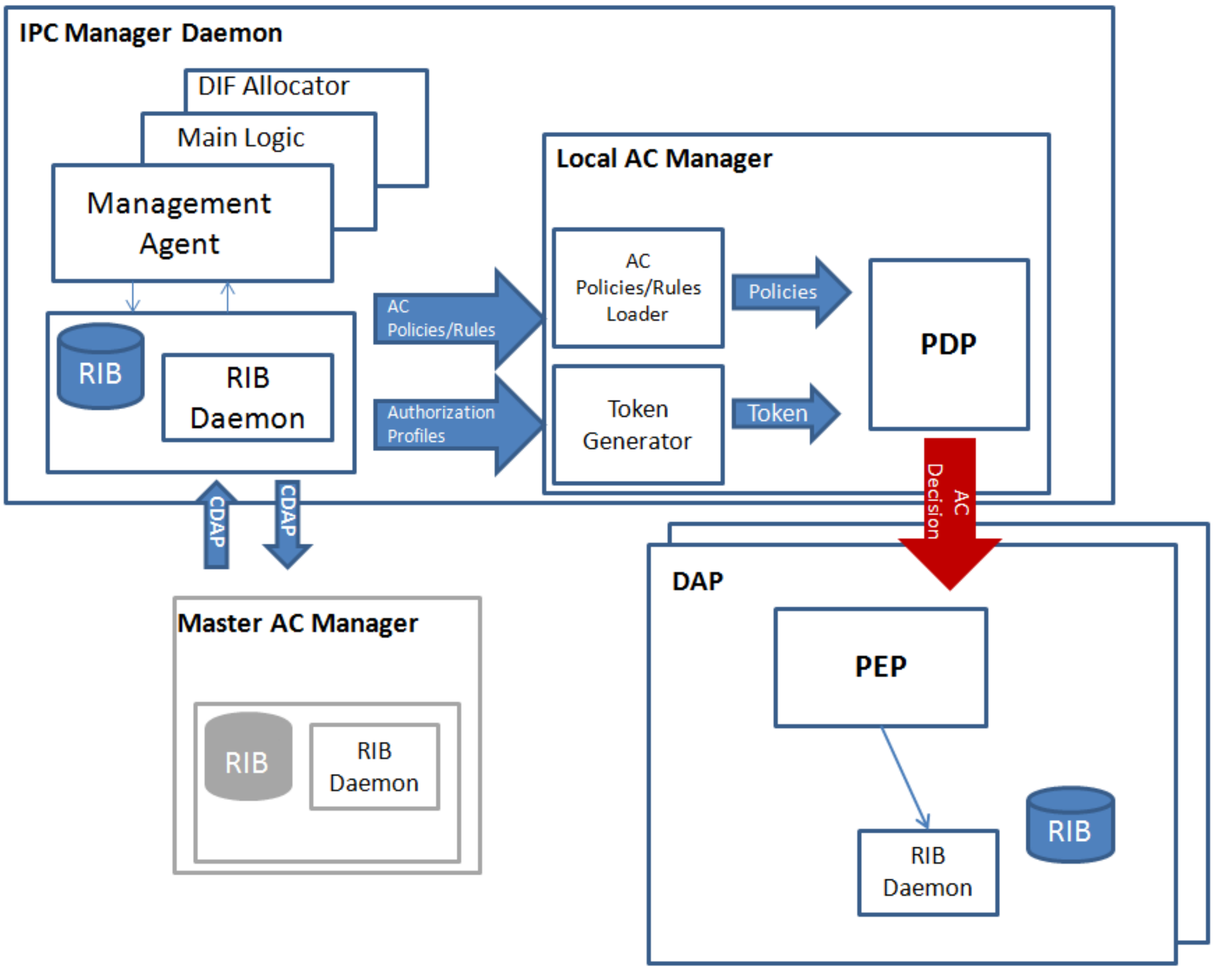

Figure 12. Inputs and outputs of the AC system

3.3. CBAC Implementation for PoC

The different AC components will be implemented in the IPC Manager block. As shown in Figure 13, “Implementation Scenario of CBAC RINA for component-level PoC tests”, we consider in our implementation three machines to implement a DAF Level with two DAPs DAP1 and DAP2. A separate machine will be used to run the AC Manager Master and store the system profiles, i.e. the DAP, RIB, DAF and user profiles, as described in previous sections.

Message exchange will be implemented between the and the AC Master Manager to request the needed profile on access control request basis. It is the DAF Management layer.

Message exchange will be implemented between DAP1 and DAP2 to exchange the access control requests and replies.

Specific interfaces from RINA implementation will be used (Netlink socket).

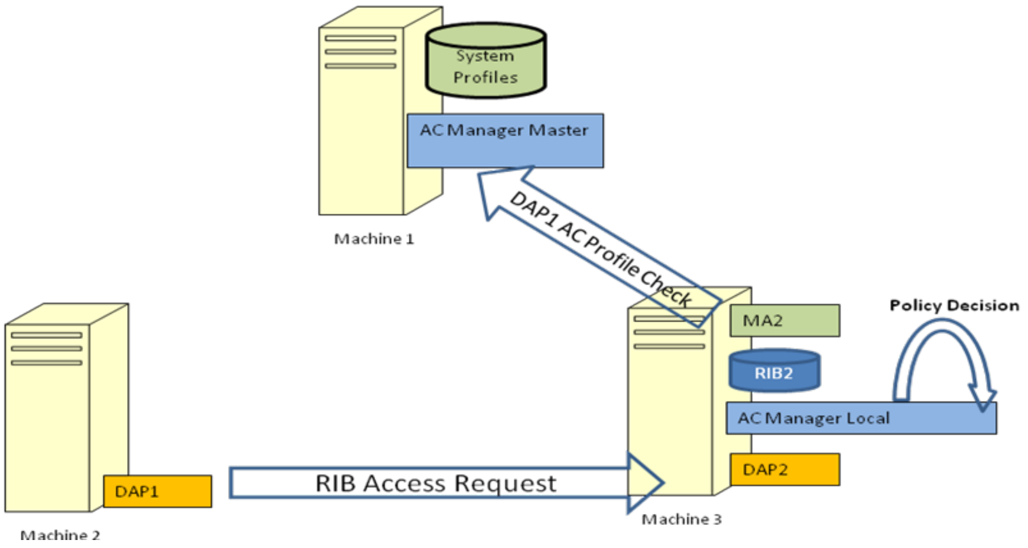

Figure 13. Implementation Scenario of CBAC RINA for component-level PoC tests

3.4. Component-level PoC Tests for Access Control

We consider two test scenarios as the initial work.

In scenario 1, DAP2 will be a streaming Video Server or a file transfer server. DAP1 will try to access the Video streaming service where the video is stored in RIB2 of DAP2. We consider DAP1 profile with a PERMIT Policy and we’ll show the start of the video streaming or file transfer service.

In Scenario2, DAP1 profile corresponds to a PERMIT Policy but for 2 minutes Streaming or file transfer, the test will show the streaming video or file transfer that stops after 2 minutes.

In our experiments we'll measure the time for access control service by making the decision by the Master AC Manager then by the Local AC Manager. We will also stress the AC Master with several requests to see how the AC system is able to handle important requests of AC.

3.5. Next Steps for CBAC Activity

The CBAC architecture has been defined to provide a complete description of the requested features. In this deliverable, detailed technical specifications and the relevant interfaces are provided. The interaction between the AC actors and internal RINA components have been provided in the sequence diagram.

In the next steps, we plan to implement different AC modules and then schedule the integration with the other components in the scope of WP6.

More precisely, important steps will be to synchronize with WP5 regarding the addition of the profiles defined here in the system profiles information base and the communication interfaces between DAP elements of the DAF and the DAF Management where the AC Master Manager interacts and see in WP6 how it is possible to integrate our proposed CBAC into RINA architecture.

4. Multi-Level Security

Multi-Level Security (MLS), as described in D4.1 [D4.1], refers the protection of data or “objects” that can be classified at various sensitivity levels, from processes or “subjects” who may be cleared at various trusted levels. A strict definition of MLS includes a formal model of classification levels for data and clearance levels for users, together with rules to prevent inappropriate access by users to data that is at a higher classification level than their clearance. Such a model is appropriate in many high assurance applications, and is often mandated in government and military contexts by policy. Such models typically make it difficult to share data effectively. However, a growing number of initiatives are aimed at situations where data sharing is a key requirement, and only moderate assurance is required. In these cases, MLS models and solutions may either be dictated by policy or are being considered to provide higher assurance than in current applications. However, such models and solutions are generally not flexible enough for the data sharing requirements.

In D4.1 [D4.1], we proposed a number of MLS architectures that enable secure data sharing to be achieved on the common RINA infrastructure. There are two components that are needed to create these MLS architectures: Communications security and Boundary Protection Components (BPC).

Communications security protects the end-to-end transfer of data between IPC/application processes. This is needed to ensure that data cannot be inappropriately read from the communication channel (e.g. via eavesdropping or accidental leakage), and that data at different classification levels is not inappropriately mixed.

To make an MLS system practical it is generally necessary to allow for at least some capability to send data from a high system to a low system, e.g. to allow higher cleared users to send emails to lower cleared users. This capability needs to be carefully controlled to prevent accidental or deliberate release of sensitive information by users or malicious code. The BPC is used to control such a flow of data, to ensure that data transferred from the high system is actually at a suitable classification level for the low system. It may also control data imported to sensitive network, e.g. check for malware.

In the remainder of this section we consider current techniques for implementing communications security and boundary protection and how these could apply to RINA. We then specify the components required to implement both communications security and boundary protection in a RINA network.

4.1. MLS Scenarios

4.1.1. MLS Communications Security

Communications security enables sensitive data to be sent over untrusted network by cryptographically protecting the confidentiality and integrity of data. This ensures that the data cannot be inappropriately read from the communication channel and that data at different classification levels is not inappropriately mixed. It also includes authentication of the end points to ensure that they are suitable for accepting the data being communicated, based on its classification level.

Communication solutions in current networks can be characterised by the layer of the Open Systems Interconnection (OSI) stack at which they operate, as described in D4.1 [D4.1], and whether they are so-called "bump in the wire" or "bump in the stack" [RFC4301] solutions. "Bump in the wire" solutions are hardware devices designed to sit between an end device and an untrusted network. As these are bespoke solutions built from scratch to provide communications security (and nothing else), they can be produced to very high levels of assurance. However, the additional devices required can be expensive and take up space. "Bump in the stack" solutions are generally software solutions designed to integrate into existing end devices. The assurance achievable in these is fundamentally limited by the device and the software into which they are integrated, however, they do not take up additional physical space and can be a lot cheaper. In addition, the assurance achievable can be enough for many commercial and less stringent defence and government situations.

4.1.2. Boundary Protection Component

The Communications Security component described above protects sensitive data from being inappropriately accessed by separating data at different classifications. However, an MLS network using only communications security is very constrained, as it very hard to share data between systems at different levels. The only means of sharing data is via manual transfer. For example, if a user on a High system wishes to share some data with a user who only has access to a Low system, the only way this is possible is for the High user to manually enter it into the Low system. If they needed to send the same information to multiple users at multiple levels, they would have to replicate this action for each level.

Therefore to make an MLS system practical it is generally necessary to allow for at least some "write down" capability, i.e. some means of enabling data sharing between systems at different classifications. For example, this would allow higher cleared users to share data that is no longer considered sensitive or that has had its sensitive parts removed with lower cleared users. Clearly, this “write down” facility needs to be carefully controlled to prevent accidental or deliberate release of sensitive information by users or malicious code, and this is where “Trusted Downgrade” and “Boundary Protection Component” (BPC or “Guard”) products are used.

Trusted Downgrade is typically a facility provided within MLS operating systems that allows highly trusted users, and perhaps applications, to modify the labels on data in special cases. This facility would typically be protected to high assurance levels so that the risk of malicious code exploiting it is very low.

Where formal, and trusted, labelling is not present (i.e. in most MLS approaches described in D4.1), there is no Trusted Downgrade as such, but the ability to make data available from higher classified systems to lower classified systems is often required. BPCs are used to control such an information exchange, to ensure that data transferred from the high system is actually at a suitable classification level for the low system. They provide assured data flow between networks of differing sensitivity, enabling Low classified data residing on a High classified system to be moved to another Low classified system.

There are five main methods of boundary protection used to prevent accidental or deliberate release of sensitive information: manual transfer, label checking, deep content inspection, content modification and user-sanctioned export. Note that although some of these methods have similar functionality to a firewall, the difference is that a BPC is an assured solution that must be effective in providing control over information exchange even when under attack or when it fails.

Manual transfer requires a person to check the true classification level of the data to be transferred, and to re-enter the data (perhaps suitably sanitised) into the low classification system manually. Clearly, this is a costly and inefficient solution. It is also subject to human error, depending on how complex the data is.

Although formal, and trusted, labels may not exist, other, informal, labels may be used to check the content. Examples of labels include simple text strings, such as classification statements in Microsoft Word document headers, or slightly more structured labelling of Word documents as provided by Purple Penelope [Gollmann] Where such labels exist, a BPC can simply search for them and ensure that release rules are adhered to. For example, DeepSecure XML Guard [DeepSec] uses embedded security labels within XML data objects. This can be effective against accidental release of sensitive data, but as the labels are not trustworthy, users or malicious code could deliberately mislabel data to bypass the protection. Therefore, the level of assurance provided is quite low. Such label checking approaches are also application specific, and are likely to require BPCs to be constantly updated and added to as applications are modified and new ones are added over time.

Another BPC approach uses deep content inspection, where all of the data is inspected to determine, through some knowledge of the data semantics, what its classification level is and/or that it does not contain hidden data. Techniques include keyword searching of text in e-mails or documents, or the analysis of images to detect hidden data. For example, Nexus Watchman [Nexor] determines the classification of a message based upon a weighted hit-word count of the message content. Clearly, deep content inspection is highly application-specific, with the same consequent issues as for label checking. In addition, the reliability of, and hence level of assurance in, such methods is generally quite low. They can be somewhat effective against accidental release of sensitive data and deliberate release of sensitive data by unsophisticated attackers or malicious code. However, more sophisticated attackers and code can generally get around the inspection, especially if they can obtain or infer the content inspection rules. As an example, consider an attacker that wishes to export a sensitive text document. The BPC may have a text keyword checker, but the attacker could bypass this by scanning the document and sending the image instead. A more sophisticated BPC may have optical character recognition (e.g. [MAGEN]), but the image could be manipulated by the attacker to make this fail (e.g. CAPTCHAs [Gollmann]). The attacker could also revert to some proprietary (to the attacker) method of encoding text in an image file, or even to hiding the text in redundant parts of a real image (steganography). A BPC that blocks all images may also not help, as the attacker could encode the text in an innocuous text document, by, for example, manipulating white space [Mansor]. Essentially, there is an arms race with the attacker having almost limitless ways to defeat content inspection mechanisms as they are developed, and there is no “silver bullet” technical solution here. A final issue is that these approaches are processor intensive and can add a delay into the release of data. This can be particularly problematic for large volumes of and/or real-time data, such as video streaming.

Content modification aims to modify content to remove potential ways in which sensitive data can be leaked within it. Generally, these techniques concentrate on the protocols used to transport the data, rather than the data itself, and are aimed at limiting or eliminating the possibility of covert channels. In other words, content modification is applied to situations in which the data itself is perfectly legitimate and releasable, but an attacker or malicious code is using manipulation of the transport protocol to sneak data through a BPC (see QinetiQ Sybard® ICA Guard [Sybard]). A "protocol break" BPC is a common approach, where the BPC acts as a proxy for the protocol. It will terminate the protocol and re-encode protocol messages according to its own rules and interpretation of the original message. It may also manipulate the timing and/or size of protocol messages (e.g. adding delays, padding or even sending "dummy" messages) to protect against these potential covert channels ([Zhiyong] for example). Such approaches are quite effective against use of the protocols to leak data, even by sophisticated attackers and malicious code, but of course do not prevent the payload data itself being used to leak data (the problems associated with deep content inspection as detailed above still apply). Protocol break BPCs can be very effective in protecting the integrity of the high system from messages sent from the low system. In particular, malformed protocol messages and buffer overflows can be effectively stripped out by this approach (both of which are very common forms of attack).

User-sanctioned export abandons the idea of the BPC doing any checking of the data. Instead, it simply makes sure that an end user has to authorise its release, and that this fact is securely recorded in a way that cannot be repudiated by this user at a later date. The aim is to place the onus on the user to check the data, and to act as a deterrent to the user accidentally or deliberately releasing information they know they should not, or are just not sure of the provenance of. Of course, this cannot prevent the release of such data, but aims to make it less likely by using the threat of future legal or disciplinary action against the source of an identified leak. Its main advantages are that it is a generic approach suitable (in theory at least – see below) for any data and application, and that it is quite effective against malicious code as it guarantees that a real user is involved and not code masquerading as one. However, more sophisticated malicious code that is able to "piggy-back" onto legitimate user communications cannot easily be stopped. Even if the user is able to see and check the data the BPC receives, through some sort of trusted channel, it may be hard or impossible for them to check for modifications made by malicious code (e.g. hidden data). In addition, machine-to-machine communications cannot be supported, and in practice many types of data flow are impractical with this approach. An example is voice data, as, although it is possible for a user to sanction the setup of a VoIP session, it is impractical for them to sanction the release of each voice data packet.

Note that a special form of boundary protection can be provided by one-way data diodes. This allows data to flow from a low to a high system and prevents any possible covert channels in the opposite direction. Of course, this does not allow "write down", but can be useful in some cases to allow a more automated flow of information into a high system. Such diodes can be produced to very high levels of assurance [LinkDD], but in practice can only be used to mirror data from low to high systems rather than allowing any kind of application to application transfer.

Some BPC approaches may require the decryption of protected content to allow checking at the boundary, but this isn't ideal as it complicates key management and introduces a point of vulnerability. An alternative is to do all checking before the content is encapsulated, and then filter at the boundary on the metadata/labels. But this may be complicated and expensive to do as it needs to be replicated at all places that create content, and is also likely to be less assured as it spreads the security controls out to all application locations rather than in one highly assured BPC. Essentially, the production of metadata/labels now needs to be highly assured, but this is done by users and applications that are difficult to assure.

4.2. Achieving MLS Communications Security in RINA

For a RINA MLS network, several approaches to communications security are possible. Communications security could be applied by the application itself; alternatively, the "bump in the stack" or "bump in the wire" approaches could be used. Examples of these three approaches are discussed below. In each example we consider an MLS network as shown in Figure 14, "Example MLS scenario", with data at two classification levels: High and Low. Each Application Process (AP) and IPC Process (IPCP) is cleared to access data at either High (shown as red in the figures); or Low (shown as green in the figures). Each DIF and DAF has a classification level of either High or Low. IPCPs and APs are only able to enrol in a DAF or DIF for which they have the appropriate clearance level, i.e. an IPCP cleared to High can only enrol in a DIF classified at High and an IPCP cleared to Low can only enrol in a DIF classified at Low. The following examples only consider a single AP in each system. However, in practice, multiple APs could use the same IPCP in the underlying DIF to send their data. In the following diagrams, a black box labelled "Z" is used to show where the communications security is applied when sending the PDU and removed when receiving the PDU.

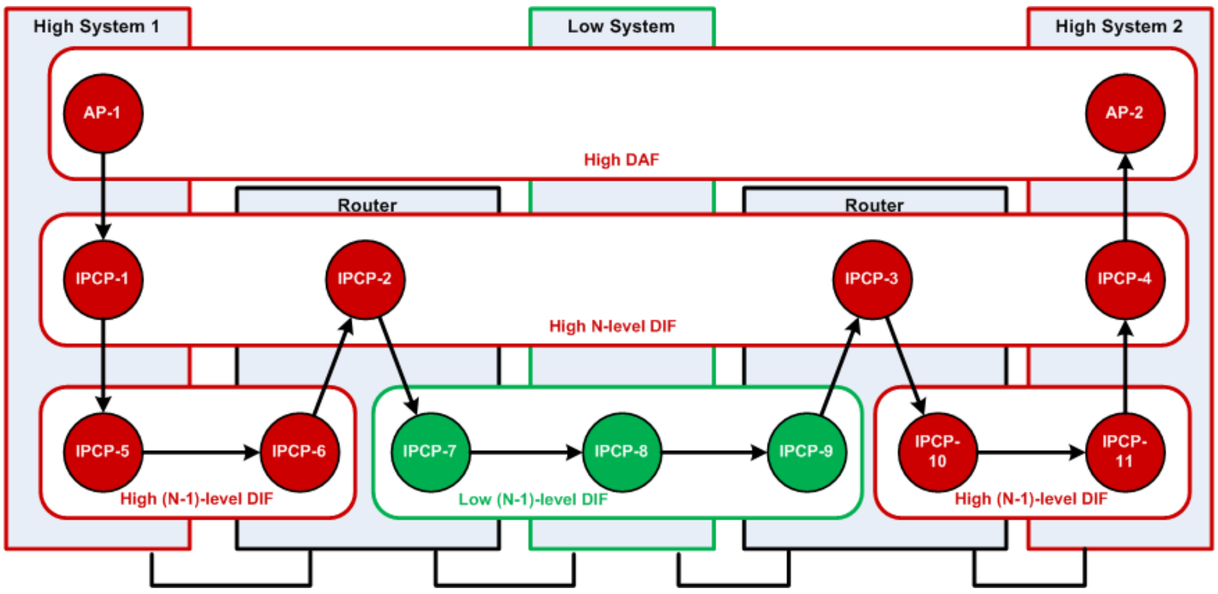

Figure 14. Example MLS scenario

4.2.1. Application-level

Communications Security can be implemented in the applications (AP-1 and AP-2 in Figure 14, "Example MLS scenario"). AP-1 encrypts the application data before it is packaged into SDUs to be sent over RINA. The SDU remains encrypted while it is sent over the RINA network. Once it has been received at the destination application (AP-2), it is decrypted. This allows fine-grained protection to be applied to the data, i.e. protection can be applied to just the data that is classified as High and any data that is Low can be sent in the clear. If multiple APs in High System 1 were to send data via IPCP-1, the data from each AP would be protected with different keys and hence be cryptographically separated even if the N-1 DIF aggregates SDUs before relaying them. Since this option is implemented at the application, it does not rely on RINA to protect the data; the data is sent as if it were plaintext data.

4.2.2. Bump in the Stack

Communications security can be implemented in RINA as a "bump in the stack" solution where the cryptographic protection is applied in the end device, i.e. the system that is sending the data. There are two options for applying protection: it can be applied at the DAF, as shown in Figure 15, ""Bump in the stack" at the DAF" or at the N-level DIF, as shown in Figure 16, ""Bump in the stack" at the DIF".

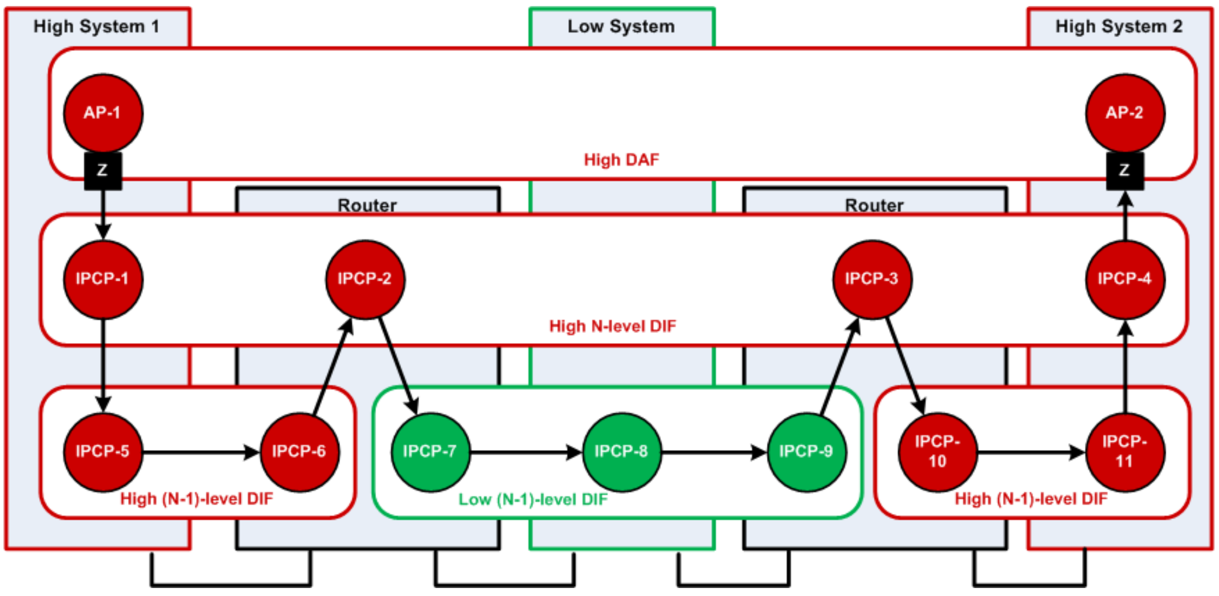

Figure 15. "Bump in the stack" at the DAF

In the "bump in the stack" at the DAF architecture, shown in Figure 15, ""Bump in the stack" at the DAF", SDUs are protected by the sending application process (AP-1) before passing it to IPCP-1 in the N-level DIF. This has the advantage that data from multiple APs sent over the same DIF will be protected with different security parameters and so will be cryptographically separated.

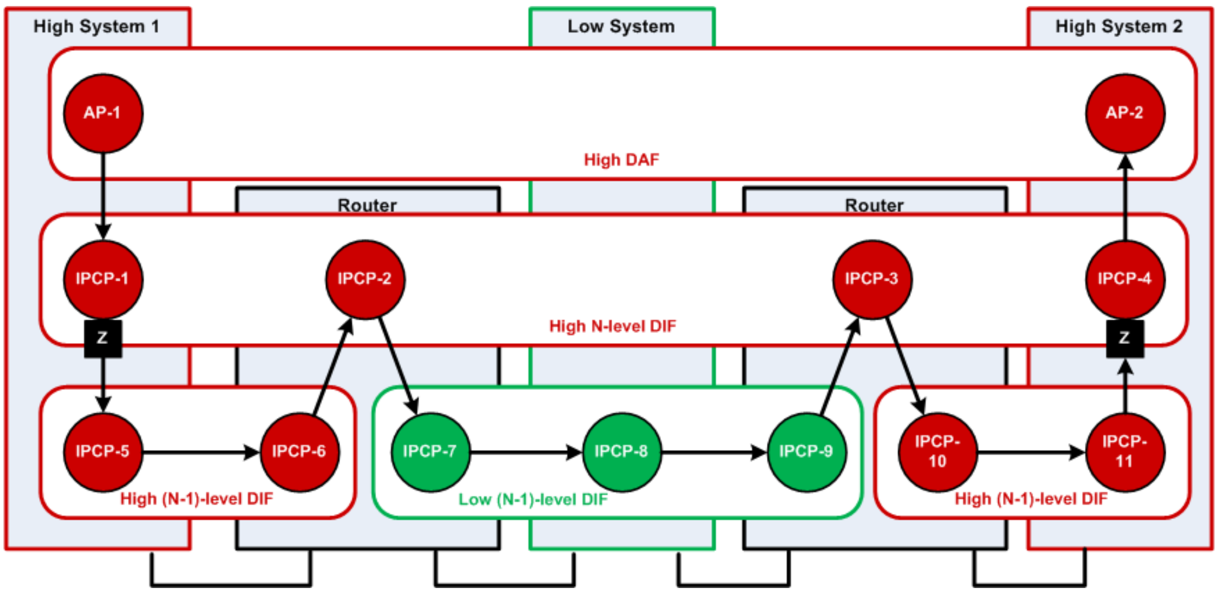

Figure 16. "Bump in the stack" at the DIF

Alternatively, the protection can be applied as "bump in the stack" at the N-level DIF, shown in Figure 16, ""Bump in the stack" at the DIF". In this option, AP-1 transfers the SDU to the underlying IPCP (IPCP-1) in the clear and IPCP-1 applies protection to the SDU before sending it to IPCP-5. Both options would have the same effect of protecting the SDU end to end from the sending High System to the receiving High

System. However, in this latter option, SDUs sent from multiple APs on High System 1 will be protected using the same security parameters by IPCP-1 if they are sent over the same flow and so data from different applications may not be separated. Therefore, this option is more scalable in terms of processing, as all application flows can be protected using the same IPCP flow. However, there is no specific protection for each of the individual application flows using the same IPCP.

Both of these options can be implemented using a SDU Protection policy that cryptographically protects every outgoing SDU. The specification of the SDU Protection Module and how it fits in RINA, as well as examples of SDU Protection policies for encrypting SDUs are considered in Section 5.

4.2.3. Bump in the Wire

When data classified at High is sent over DIFs that are also classified at High, the SDUs do not need to be protected. This is because the network is trusted and all IPCPs receiving the data are cleared to read it. However, if High application data is sent over a DIF classified at Low, it needs to be protected to ensure that it is not mixed with Low data and that it cannot be read by application processes that are not cleared to access it.

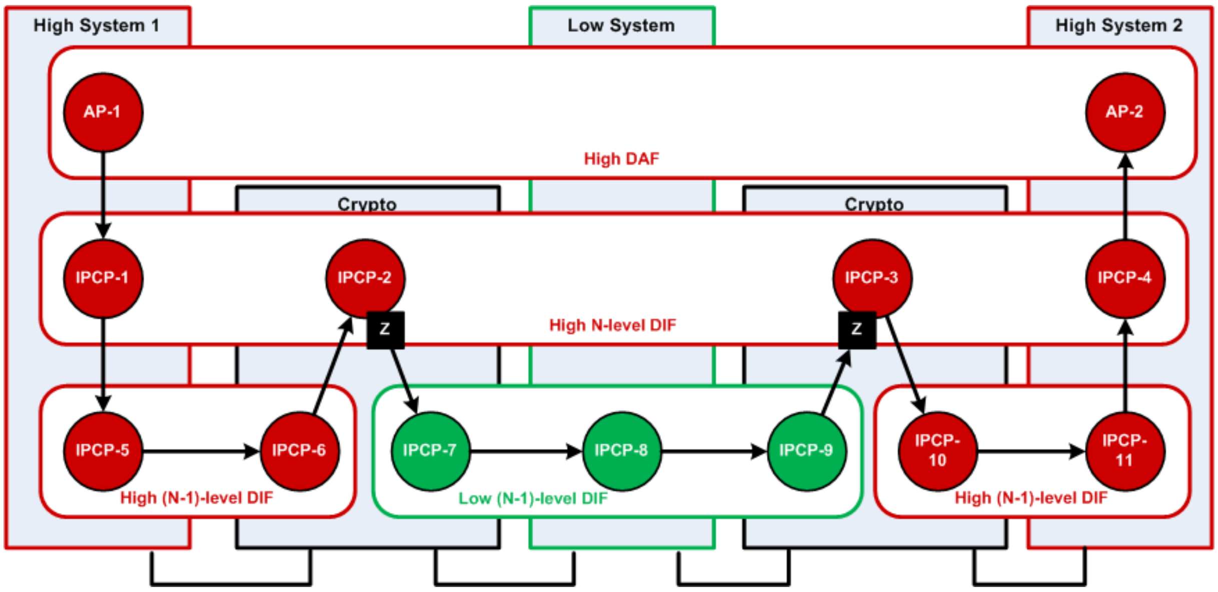

Figure 17. "Bump in the wire" solution

In the scenario shown in Figure 17, ""Bump in the wire" solution", AP-1 sends the SDU to IPCP-1, which then forwards it to IPCP-2 via IPCP-5 and IPCP-6. Since all of these IPCPs are cleared to the same level, the SDU does not need to be encrypted. IPCP-2 then forwards to SDU to IPCP-3. Although IPCP-3 is cleared to High, the underlying DIF that will transport the SDU is only cleared to Low and is therefore untrusted. Consequently, IPCP-2 must encrypt the SDU before sending it over the Low N-1-level DIF. IPCP-3 can decrypt the SDU before sending it to IPCP-4, as the N-1 DIF is classified at High. In this way, the SDU is only protected where it is sent over an untrusted DIF, which prevents multiple layers on encryption being unnecessarily applied to the SDU. It also means that only nodes that have IPCPs at multiple levels need to apply protection to SDUs. Here, protection at the IPCP flow level is more scalable, as fewer instances of IPCPs are involved in applying protection, which reduces both the processing cost and the amount of security parameters exchanged. However, it has the associated cost of losing protection at application flow granularity.

Achieving this “bump in the wire” communications security scenario requires policies for Authentication and SDU Protection. An authentication policy is needed to ensure that IPCPs only enrol in DIFs that they are cleared to, e.g. an IPCP cleared to Low cannot enrol in a DIF classified at High. This ensures that all IPCPs enrolled in a DIF are cleared to the same level and means that the clearance level of an IPCP can be inferred from the DIF in which it is enrolled. Therefore once an IPCP has enrolled in a DIF, it can communicate with any IPCPs in the same DIF without needing to verify their clearance level.

The Authentication policy is also needed by the SDU Protection Module to negotiate security parameters for the flow, e.g. the cryptographic algorithms, session keys, which are stored in the security context. The same security parameters are used for all SDUs sent over the same flow, e.g. sent from IPCP-2 to IPCP-3 in Figure 17, ““Bump in the wire” solution”. Several of the authentication policies described in D4.1 would be suitable here. For example, AuthNPassword could be used where only IPCPs that are cleared to High have a valid password for enrolling in a High DIF. Section 2 specifies the Authentication Module and example authentication policies that could be used in an implementation of “bump in the wire” communications security.

To implement the “bump in the wire” configuration, a cryptographic SDU Protection policy is needed to encrypt PDUs before they are sent over an untrusted DIF. The policy should only encrypt SDUs sent over flows through an underlying DIF that is at a lower classification level; flows through an underlying DIF at the same classification level should be left in the clear. There are two ways that this could be achieved. The first is to use the Manager and Management Agent in the Distributed Management System (DMS), described in D5.1 [D5.1], to configure the SDU Protection policy for each flow. Each time a new flow is established from a High DIF to a Low DIF, the Manager configures the SDU Protection policy to encrypt SDUs sent over the flow. Alternatively, a customised SDU Protection Policy could be used that can decide whether to apply encryption to a PDU based on the classification of both the PDU and the flow. This latter option will specified below.

4.2.4. Specification and Design of the Bump in the Wire Solution

Here we specify the SDU Protection policy, which we call the 'MLS Encryption Policy', needed to implement the "bump in the wire" MLS architecture shown in Figure 17, ""Bump in the wire" solution". The policy is implemented in the SDU Protection Module of IPCPs that apply protection to and remove protection from SDUs that are sent over an untrusted underlying DIF, e.g. IPCP-2 and IPCP-3 in Figure 17, ""Bump in the wire" solution".

Figure 18, "Block diagram of how MLS encryption policy fits in RINA" illustrates how the custom MLS Encryption Policy fits within the RINA IPCP. The RINA components involved are the SDU Protection Module, RMT and the Authentication Module.

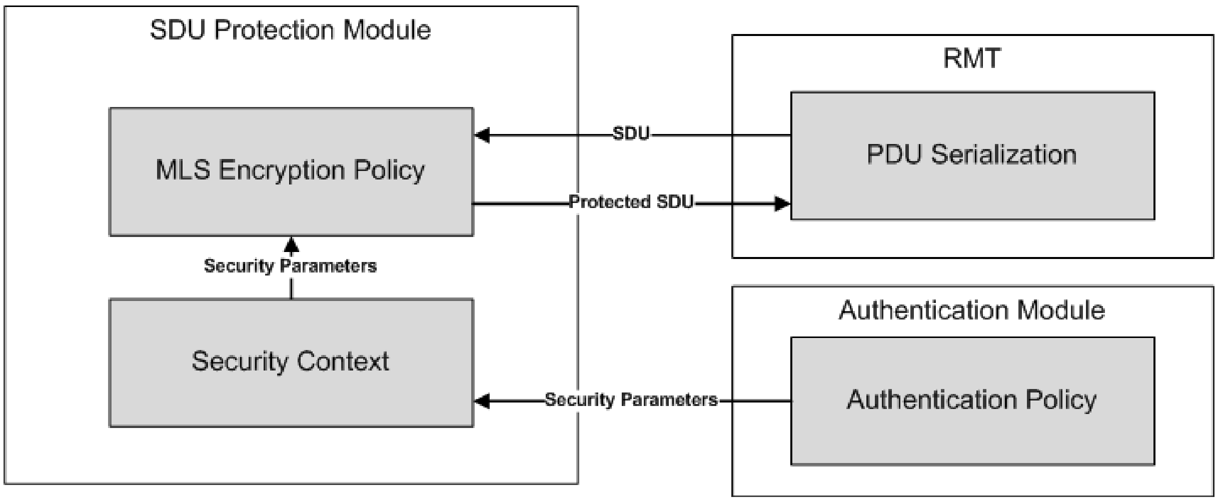

Figure 18. Block diagram of how MLS encryption policy fits in RINA

During the enrolment process, the Authentication Module, described in Section 2, authenticates the IPC process joining the DIF. Only IPCPs that successfully authenticate can enrol in the DIF. Its Authentication Policy defines the authentication mechanism used to authenticate the joining IPCP. It also updates the SDU Protection Module's Security Context with any security parameters, e.g. key material and cryptographic algorithms, which may be negotiated as part of the authentication process. These security parameters are negotiated per flow, so that an IPCP has a different set of keys for each IPCP within the DIF. The security parameters are not tied to the Application Process sending the SDUs, so that SDU s belonging to different APs sent over the same flow will use the same security parameters.

When a PDU is to be sent from this IPCP to the underlying flow, RMT passes PDUs from DTP instances to the appropriate (N-1)-ports. Its serialisation task invokes the SDU Protection Module, described in Section 5, which applies protection to outgoing PDUs according to its SDU Protection policy. MLS Encryption Policy is an SDU Protection

Policy that implements the “bump in the wire” Communications Security scenario described above. It applies encryption to outgoing PDUs that are to be sent over a flow at a lower classification level. The Security Context contains the configuration data and security parameters needed by the SDU Protection policy, e.g. the encryption key and encryption algorithm to apply.

4.2.5. Interaction of Components with SDU Protection Policy

Figure 19, “Sequence diagram showing the interactions when the SDU is sent over an untrusted underlying DIF” shows the sequence of interactions between the RINA components when applying the MLS Encryption policy to an SDU being sent over an untrusted DIF.

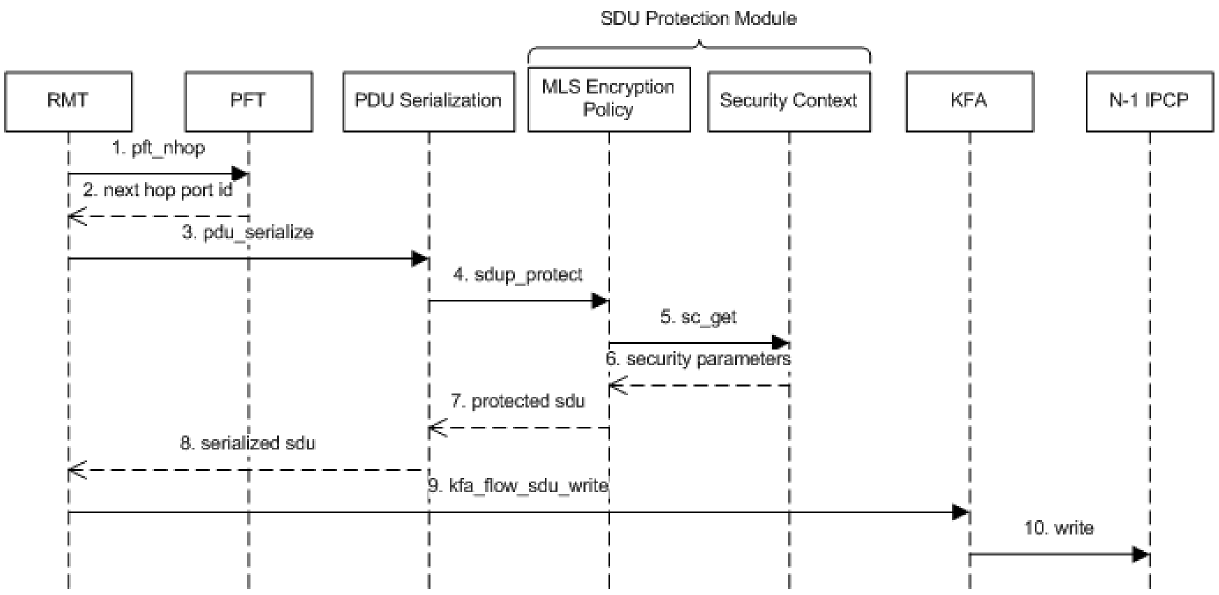

Figure 19. Sequence diagram showing the interactions when the SDU is sent over an untrusted underlying DIF

1. When an SDU is to be written to the underlying flow, it is passed to RMT. RMT looks up the port to be used to send the PDU to the destination address in the PDU Forwarding Table (PFT) via pft_nhop.
2. The PFT returns the port ID of the next hop
3. RMT sends the PDU to be serialised by calling pdu_serialize
4. PDU Serialization then invokes the SDU Protection Module, which applies the MLS Encryption policy. This policy determines that the PDU needs to be protected, as it is to be sent over an untrusted DIF
5. The MLS Encryption policy obtains the necessary security parameters, e.g. the session encryption key and encryption algorithm, from the Security Context that is established during authentication.

6. The Security Context returns the security parameters for the flow that the PDU will be sent over.
7. The MLS Encryption Policy applies protection to the PDU using the security parameters
8. The serialized and protected PDU is then returned to RMT.
9. RMT then sends the PDU to the KFA
10. The KFA write the PDU to the outgoing port to be passed to the underlying IPCP

Figure 20, "Sequence diagram showing the interactions when the SDU is sent over a trusted underlying DIF" shows the sequence of interactions between the RINA components when applying the MLS Encryption policy to an SDU being sent over a trusted DIF.

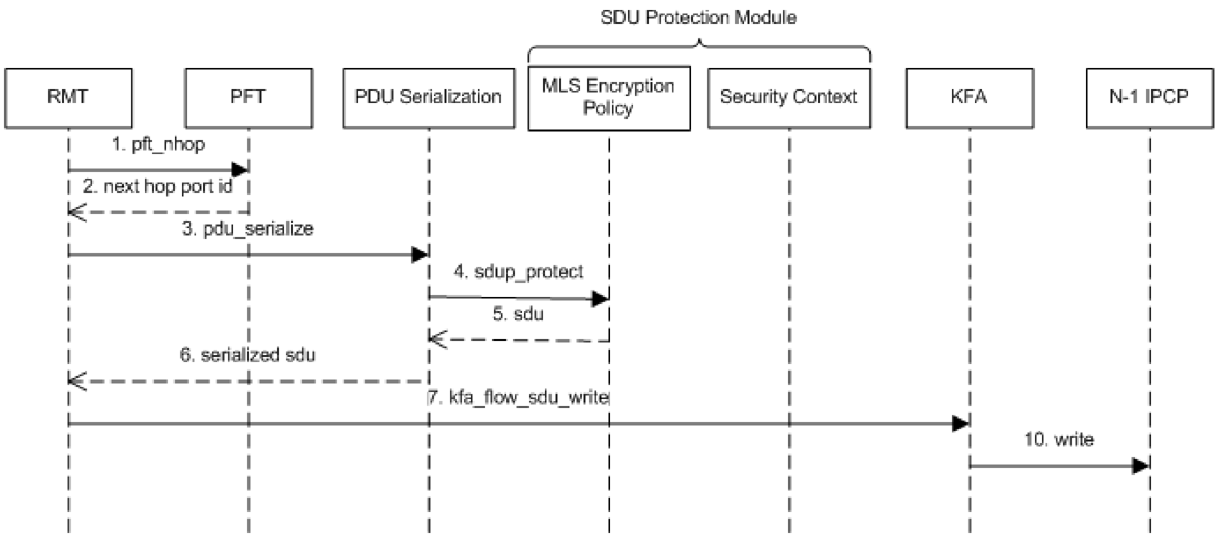

Figure 20. Sequence diagram showing the interactions when the SDU is sent over a trusted underlying DIF

1. When an SDU is to be written to the underlying flow, it is passed to RMT. RMT looks up the port to be used to send the PDU to the destination address in the PDU Forwarding Table (PFT) via pft_nhop.
2. The PFT returns the port ID of the next hop
3. RMT sends the PDU to be serialised by calling pdu_serialize
4. PDU Serialization then invokes the SDU Protection Module, which applies the MLS Encryption policy. This policy determines that the PDU does not need to be protected, as it to be sent over a trusted underlying flow
5. The SDU without protection is returned to PDU Serialization
6. The serialized PDU is then returned to RMT.

7. RMT then sends the PDU to the KFA
8. The KFA write the PDU to the outgoing port to be passed to the underlying IPCP

Figure 21, "Sequence diagram showing the interactions when the SDU is received from an underlying DIF" shows the sequence of interactions between the RINA components when applying the MLS Encryption policy when an SDU is received from an untrusted DIF and forwarded over a trusted DIF. The SDU received from the N-1 DIF is decrypted before being forwarded over the trusted DIF in the clear (i.e. without encryption).

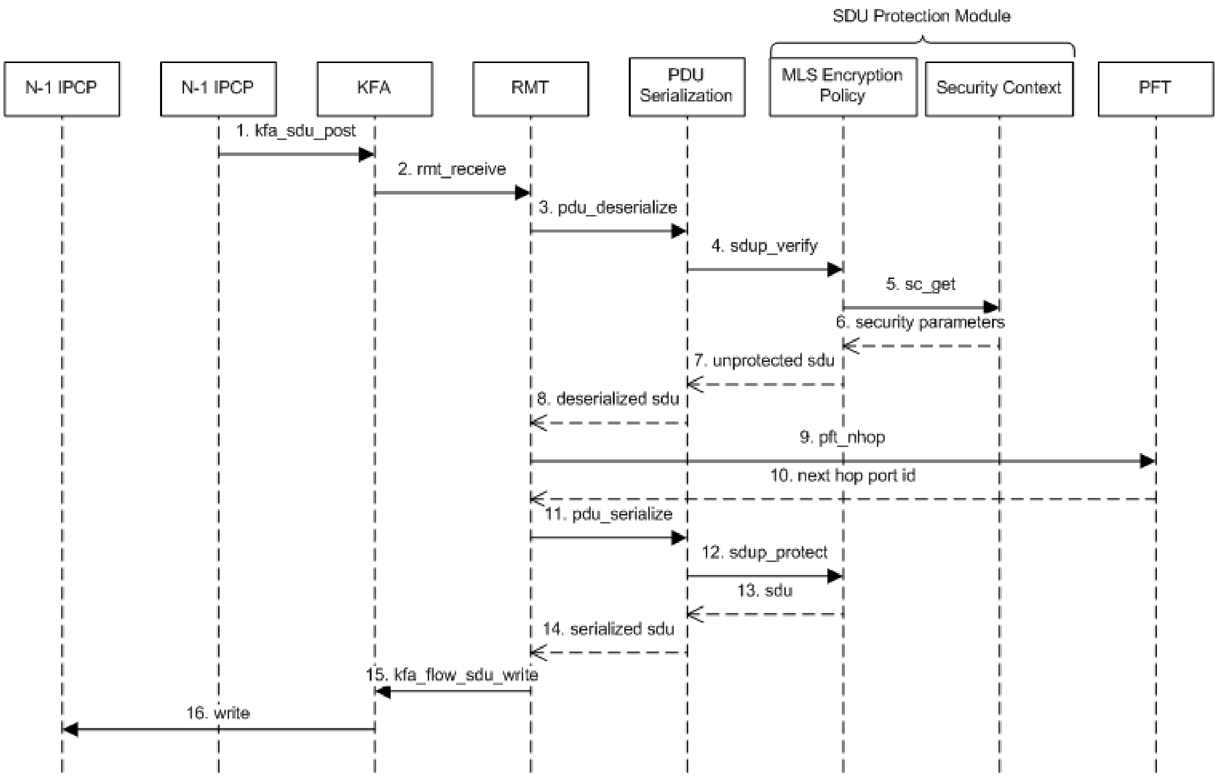

Figure 21. Sequence diagram showing the interactions when the SDU is received from an underlying DIF

1. When an SDU is received by the underlying flow, the N-1 IPCP identifies the port to which the SDU should be forwarded and calls the KFA to send the SDU
2. The KFA posts the SDU to the RMT instance associated with the flow by calling rmt_receive
3. RMT sends the PDU to be deserialised by calling pdu_deserialize
4. PDU Serialization then invokes the SDU Protection Module by calling sdup_verify, which applies the MLS Encryption policy.
5. The MLS Encryption policy obtains the necessary security parameters, e.g. the session encryption key and encryption algorithm, from the Security Context that is established during authentication.

6. The Security Context returns the security parameters for the flow from which the PDU was received.
7. The MLS Encryption policy uses the security parameters to verify and remove the protection from the SDU, e.g. to decrypt it
8. The deserialized and decrypted PDU is then returned to RMT.
9. RMT looks up the port to be used to send the PDU to the destination address in the PDU Forwarding Table (PFT) via pft_nhop.
10. The PFT returns the port ID of the next hop
11. RMT sends the PDU to be serialised by calling pdu_serialize
12. PDU Serialization then invokes the SDU Protection Module, which applies the MLS Encryption policy. This policy determines that the PDU does not need to be protected, as it to be sent over a trusted underlying flow
13. The SDU without protection is returned to PDU Serialization
14. The serialized PDU is then returned to RMT.
15. RMT then sends the PDU to the KFA
16. The KFA writes the PDU to the outgoing port to be passed to the underlying IPCP

4.3. Achieving BPC in RINA

Two options for achieving the BPC functionality in RINA have been identified. These options will be discussed in the next version of this deliverable.

4.4. MLS Implementation for PoC

4.4.1. Communications Security

The IRATI stack, described in D2.3 [D2.3] is an implementation of the RINA IPC model for a Linux-based Operating System. The functionalities of the IPC Process have been partitioned between the user and kernel spaces in order to enable the prototype to achieve and adequate level of performance and functionality. The shim IPC Processes and the data transfer and data transfer control parts of the IPC Process are implemented in kernel space, while the layer management functions of the IPC Process and the local IPC Manager are implemented in user space.

The software architecture of the SDU Protection Module and how it fits into the IRATI stack is described in Section 5. The MLS Encryption policy specified in Section 4.1 will be implemented as an SDU Protection policy and integrated with the SDU Protection Module in the IRATI stack.

4.4.2. Boundary Protection Component

The BPC PoC implementation will be discussed in the next version of this deliverable.

4.5. Component-Level PoC Tests for MLS

4.5.1. Test Environment

The MLS test environment consists of a Debian-based virtual machine (VM) image with the latest stable build of the IRATI stack installed. The VM image is hosted in VirtualBox, which is running on a Windows machine.

4.5.2. Tests to be Performed

Testing of the implementations will focus on component-level verification of the MLS Encryption Policy and the BPC. These tests aim to evaluate whether or not the implementations of the MLS components operate without error and according to their specifications. This is to prove the correct functionality of the implementation. The following tests will be performed to verify the implementation.

Table 1. Verification test of MLS Encryption policy

Test Identifier: SUITE_MLS/TRT/Crypto/1	
Type of Test	Component-level Functionality Verification
Version	1.0
Reference to Requirements	D2.1 [D2.1], Section 3.2 - security
Test Summary: This test is for assessing the functionality of the Communications Security component when data classified at High is sent over a DIF classified at Low. *Objectives:* To verify that the SDUs are encrypted when sending data over an untrusted network	
Experimentation Environment: *Test location*: MLS Testbed. *Topology*: see Figure 17, ““Bump in the wire” solution” *Traffic Load*: User traffic will be produced by Traffic Generators.	

Other RINA components used: SDU Protection Module
Test Procedure: *Initial Conditions*: • Controlled variables: controlled sending of data classified at High • Uncontrolled variables: N/A *Checks to be performed in the test*: • Verify that the data is successfully encrypted by the MLS policy at IPCP-2 • Verify that the data is successfully decrypted by the MLS policy at IPCP-3
Verdict Criteria: *Expected results*: • The data must be encrypted by IPCP-2 prior to sending it over the DIF classified at Low. • The data must be decrypted by IPCP-3. *Metrics*: N/A
Results/Comments: N/A

Table 2. Verification test of the BPC functionality

Test Identifier: SUITE_MLS/TRT/BPC/1	
Type of Test	Component-level Functionality Verification
Version	1.0
Reference to Requirements	D2.1 [D2.1], Section 3.2 - security
Test Summary:	

This test is to verify the functionality of the BPC component when data classified at High is sent to an application classified at Low. *Objectives:* To verify that only the SDUs containing sensitive data are blocked by the BPC
Experimentation Environment: *Test location*: MLS Testbed. *Topology*: see ??? *Traffic Load*: User traffic will be produced by the two application processes *Other RINA components used*: all - the BPC application under test will run over a RINA network
Test Procedure: *Initial Conditions*: • Controlled variables: classification of the data sent • Uncontrolled variables: N/A *Checks to be performed in the test*: • Verify that data classified at High is blocked • Verify that the data classified at Low is forwarded to the Low application
Verdict Criteria: *Expected results*: • Data sent from the High application to the Low application that is classified at High should be blocked by the BPC. • Data sent from the High application to the Low application that is classified at Low should be forwarded by the BPC. *Metrics*: N/A
Results/Comments:

N/A

4.6. Next Steps for MLS Activities

This deliverable defines the two components needed to achieve an MLS architecture in RINA: communications security to protect the end-to-end transfer of data between IPC/Application Processes; and a boundary protection component to provide assured data flow between IPC/Application Processes of differing sensitivity. Detailed technical specifications of both components and how they fit in the RINA architecture are provided. The interactions between the MLS components and RINA components have been defined in sequence diagrams.

The next step for the Communications Security component is to implement the MLS Encryption policy according to the specification of the SDU Protection Module in Section 5. The policy will then be integrated with the SDU Protection Module implementation. Further work will also be done in WP5 to investigate how the Manager and Management Agent can be used to configure RINA components, e.g. the SDU Protection Module, when setting up Communications Security in an MLS network. Strategies for the Manager that enable to network to be automatically configured will be defined.

The next step for the Boundary Protection Component is to implement the BPC at the DAF-level as described in Section 4.4.2. Two applications that send and receive data over RINA will also be implemented. The BPC implementation and two applications will then be integrated with the RINA network installed on the TRT testbed, described in Section 4.5.1.

5. Cryptographic Functions and Enablers

The SDU Protection module is a part of the IPC Process (IPCP) data path. The SDU Protection function is executed before the SDU is handed over to the underlying IPCP. When data are handled between IPCPs of different DIFs, SDU Protection is applied. It is intended to apply selected protective mechanisms to outgoing SDUs at the sending side and check incoming SDU at the receiving side. This is the last or the first operation applied, respectively. It aims to provide a level of protection depending on the applied policy. All the functionality of SDU protection is represented as a policy. Thus there is not a predefined common mechanism. SDU protection performs a transformation from SDU to protected SDU when the SDU is sent from the IPCP. It performs a transformation from protected SDU to SDU when the SDU is received by the IPCP. According to the overall RINA specifications, SDU protection can perform variety of functions, namely: i) lifetime limiting, ii) error checking, iii) data integrity protection, iv) data encryption, but also data compression or other two-way manipulations that may depend on the N-1 flow used. SDU Protection depends on a policy that is specific to each (N-1)-flow. SDU Protection can be used to create a secure channel between two IPCPs, though it is not excluded that SDU Protection may apply the same policy to all (N-1) flows thus creating shared security for whole N-DIF.

It is important to highlight that a DIF uses SDU protection to protect itself from untrusted N-1 DIFs (distributed applications -DAFs- that really care about protection should use their own SDU Protection policies). Securing communications in RINA is implemented via the SDU protection module. As its name suggests, the security is applied to Service Data Units (SDU). The SDU denotes a data block that is exchanged between IPCPs on a single RINA node. This follows the idea that DIFs are network areas that are independent of other possible DIFs.

A SDU is a unit of data that has been passed down from an IPCP to a lower IPCP and that has not yet been encapsulated into a protocol data unit (PDU) by the lower layer. It is a set of data that is sent by a user of the services of a given layer, and is transmitted semantically unchanged to a peer service user.

SDU protection is the part of the RINA specifications that provides functions for securing data transfer between communicating IPCPs. SDU protection is applied as the last operation on data before leaving the current IPCP. These data are packaged in SDUs. Each SDU is processed separately according to the specific SDU Protection context associated with each flow. Thus SDU protection is applied on a per-flow basis. SDU context is associate with flows to define which policy is to be applied to all SDUs of the flow. Currently, three different SDU Protection Policies are defined:

1. **Null SDU Protection** is a policy that performs no transformation - this protection mechanism is in general applicable to ShimDIFs, where protective mechanisms related to a particular communication technology or protocol are used.
2. **Basic SDU Protection** is a policy that applies fundamental protective mechanisms. These mechanisms include time life limiting (TTL) and error checking (CRC).
3. **Cryptographic SDU Protection** relies on the implementation of the following four key SDU protection mechanisms that applies to every SDU:
    - SDU Lifetime method deals with limiting maximum lifetime of each SDU to avoid its unlimited circulating in a network. As a part of this mechanism, replay detection is provided.
    - SDU Compression method specifies methods of compressing data in order to reduce the data size or to add entropy to the data when encryption is to be applied.
    - SDU Encryption method specifies which method to use for securing content by applying cryptographic encryption.
    - SDU Integrity method specifies which algorithm to use for computing cryptographic hash of the content in order to enable detection of changes of the SDU content.

Suitable methods are well known for implementing all four SDU protection mechanisms. SDU protection mechanisms define profiles that provide a particular algorithm and its possible parameters. SDU Protection is located at the boundaries of the IPCP. For each SDU, the module knows to which N-1 flows this SDU has to be written to or has been read from. It is this possible to associate SDU Protection contexts to N-1 flows. SDU is sent to underlying DIF using specified port. **The SDU protection policy proposed in this section does not assume that the underlying N-1 flow is reliable**. For this reason, protected SDUs need to carry enough additional information for receiver to successfully decrypt them.

5.1. Cryptographic Concepts used in SDU Protection Policy

This section provides a description of concepts, methods, algorithms, etc that are used in the design, specification and implementation of the SDU Protection module.

5.1.1. Replay Detection

Replay detection is implemented using a replay window mechanism as specified in [RFC2401]. Each crypto block is numbered using a sequence number to support replay detection. This sequence number must be protected by appropriate integrity mechanism. In short, replay detection works by checking duplication of SDUs and by discarding SDUs which are too old. Both of these conditions can be realized using SDU numbering.

5.1.2. Ciphering Modes

It is not possible to use stream ciphering modes for this particular encryption policy as these depend on reliable data delivery. Instead, block ciphering modes are suitable in this case. CTR encryption using a counter value is an efficient method used for creating a secure channel over an unreliable data delivery service. Algorithms such as DES, 3DES and AES can be used in this mode. There are two considerations that must be followed to apply this mode correctly:

- The same secret key and counter must not be reused for encrypting different messages
- An integrity check is necessary to protect a message from modification

5.1.3. HMAC

A Hash-based Message Authentication Code (HMAC) is a function for calculating message authentication code that involves a secret cryptographic key. HMAC is usually used for ensuring message integrity and in key derivation functions.

HMAC is defined (according to [RFC2401]) as follows:

```
HMAC(K,m) = H(K XOR opad , H(K XOR ipad , m))
```

where

- `H` is a cryptographic hash function, e.g. SHA-1,
- `K` is a secret key adjusted to block size of H (either padded or hashed),
- `m` is the message to be authenticated, and
- `opad` and `ipad` are the outer and inner padding, respectively.

5.1.4. Diffie-Hellman Key Exchange

The Diffie-Hellman (DH) method of secret key exchange is based on existence of the following equation:

```
g^ab = (X_b)^a MOD p = (X_a)^b MOD p

X_a = g^a MOD p
X_b = g^b MOD p
```

Wherein, `p` is a large prime number, `g` is generator and `a`, `b` are secret random numbers private to each party. An initiator sends message `(p,g,X_a)` to a responder, which selects its secret `b` to compute `X_b` as its response. Both parties can then compute the same shared `g^ab` secret key.

5.1.5. Keying Material

The key generation mechanism described in this section stems from an adaptation of IKE methods, as described in RFC 4306 [RFC4306]. Each party p needs three write keys, namely:

- session key used for encryption (`K_enc^p`). The size of this key depends on the cipher algorithm used. Usual values are 56bits, 64bits, 80bits, 128bits, 192bits, or 256bits.
- session key used for hashing (`K_dig^p`). The general rule is that the key length for message integrity checking should be the same as the length of the key used for message encryption.
- session key used as nonce for counter generation (`K_seq^p`). The length of this key depends on the block size of the encryption cipher used. This is because, the counter is obtained by concatenating a sequence number and counter key. Typical block sizes are 64bits, 96bits, 128bits or 192bits.

These write keys are generated from the single Master Secret Key `K_master` that needs to be provided at the initialization of the secure channel. Let `PRF(K,S)` be a pseudo-random function, e.g. based on SHA-256 algorithm, negotiated by both parties as a part of security context of the secure channel. According to IKEv2, keying material can be generated in the following way. First, shared secret `K_seed` is computed from Master Secret Key `K_master` and random generated values `N_i` and `N_j`:

```
K_seed = PRF(N_i | N_r, K_master)
```

where

- `N_i` and `N_r` are random nonce values generated by initiator and responder, respectively,
- `K_master` is a Master Secret Key that can be exchanged using DH method or can be a pre-shared key.

Note that while `K_master` must be kept secret by communicating parties, nonce values `N_i` and `N_r` may be sent as plaintext. Computed secret `K_seed` is the key derivation key used for computing a collection of session write keys using `PRF^+(K,S)` function. This computation consists of a chain of PRF function applications defined as follows:

```
PRF^+ = T_1 | T_2 | ...
```

where

- `T_1 = PRF(K,s | 0x1 )`
- `T_i+1 = PRF(K, T_i | S | 0x(i+1))`

Function `PRF^+(K,S)` generates blocks of data enjoying pseudorandom properties. These blocks thus can be used as session keys. Computing all necessary keying material is performed by applying `PRF+` function until there is enough data, which depends on the key sizes of the algorithms used for encryption, hashing and counter generation. Mapping `T_i` blocks to keys is straightforward. Each key takes as many bits from `T_i` blocks as necessary. The computation ends when all keys have assigned values. The computation of writing keys for the initiator and responder is defined as follows:

- For initiator:

```
(K_enc^i | K_dig^i | K_seq^i) = PRF^+(K_seed, N_i | N_r )
```

- For responder:

```
(K_enc^r | K_dig^r | K_seq^r) = PRF^+(K_seed, N_r | N_i )
```

Using the above defined equations both communicating parties are able to generate all the keying material knowing a common secret key and two nonces. These nonces can be generated by each party and exchanged during the connection establishment and authentication phase.

5.1.6. Counter Mode

Ciphers can be used in various ciphering modes. However, only the Counter Mode is initially considered for the proposed SDU Protection Policy. The counter mode allows for an efficient implementation that provides an efficient method for encrypting and decrypting high-speed data. It relies on the quality of the cipher and the uniqueness of the counter value. The counter value consists of a sequence number and a nonce based on a sequence key. This provides the advantage that each encrypted block is independent of other blocks, which works well if data delivery is not reliable. For reliable data transport, this mode adds a little overhead represented by the necessity to maintain a sequence number counter with specific properties - the counter must not be repeatedly used with the same key. The counter length must be equal to the block-size of the cipher algorithm used for data encryption. The method for counter computation varies with block-size. The counter is computed using the following recipe:

```
counter = K_seq | uint32(seq_num) | uint32(0x0)
```

The counter can be used with different block sizes. Current cipher suites support blocks of length 64, 96, 128 and 192.

5.1.7. Selecting algorithms for SDU Protection Policy

The designed SDU Protection policy based on cryptographic methods provides a secure communication channel that meets requirements identified in D4.1. This is achieved by combining four mechanisms for controlling PDU lifetime, offering the possibility to encrypt SDU content, protecting SDU from unauthorised modifications and reducing the size of SDU by applying compression. Encryption and integrity mechanisms secure the communication. The strength of the security measures applied depends on the combination of the methods used for encryption and integrity protection. The following table shows the possible combinations and their properties in terms of the security provided as defined in the presented SDU Protection Policy. More information on the status of individual algorithms can be found at [ngenc]. In the table, algorithms are classified into three groups:

- Avoid: algorithms that do not provide an adequate security level against modern threats. It is recommended that these algorithms should not be used in application relying on strong security requirements.

- Legacy: algorithms provide a marginal but acceptable security level. These algorithms can be used if there is not better option. For these algorithms there are techniques that help to mitigate the security problems and thus increate a level of security provided to acceptable.
- Acceptable: algorithms provide adequate security.

Table 3. Message integrity algorithms:

Algorithm	Status (possible mitigation)
MD5	avoid
HMAC-MD5	legacy
Ripemd160	legacy
SHA1	legacy (short key lifetime)
HMAC-SHA1	acceptable
SHA256	acceptable
SHA384	acceptable
SHA512	acceptable

Table 4. Message encryption algorithms:

Algorithm	Status
Aes	acceptable
Des	avoid
3Des	legacy (short key lifetime)
Rc2	avoid

The strength of the algorithm is relative to a security level expressed in bits [NIST SP 800-131].

Algorithm	Security Level
Aes-128	128
Aes-192	192
Aes-256	256
Des	56
3Des	80 (112)
Rc2	40

Algorithm	Security Level
SHA1	80
SHA256	128
SHA384	192
SHA512	256

Different classes of applications requires different levels of security. The following are different application classes:

Application Class	Minimum security level
Low	≤ 64
Medium	≤ 128
High	≤ 256
Extreme	> 256

Achieving a higher security level means performing more computations. Thus the correct application level should be considered with respect to not only security but also costs.

According to the given classification of algorithms the combination of security algorithms for integrity and encryption is classified considering the least security level provided. This means that for achieving the High security level, AES-256 and SHA512 combination should be selected.

5.2. Specification and Design of the SDU Protection Component

5.2.1. Software Architecture of the SDU Protection Component

This section provides a software architecture in block diagrams and in terms of the functions and workflows at a high-level level, specifically for SDU protection and how it works and fits into the IRATI RINA implementation. SDU Protection functions are invoked from the PDU serialization and deserialization module. Serialization/deserialization (SerDes) tasks are part of RMT that operates over PDUs. The block diagram showing the context of SDU Protection is in Figure 22, “SDU Protection Block Diagram”.

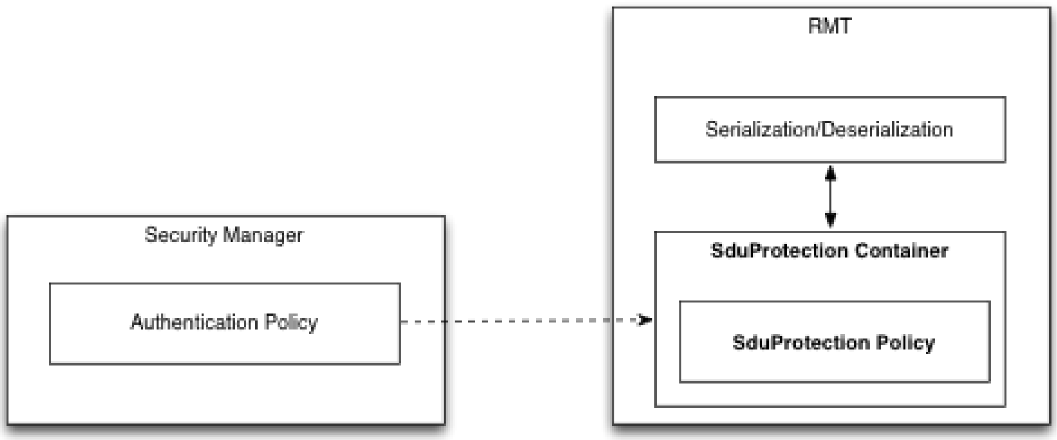

Figure 22. SDU Protection Block Diagram

SDU Protection is realized using SDU Protection Policies. Thus, to integrate into the IPCP architecture, the SDU Protection container is specified which provides an interface between RMT and the instantiated policies. Also this container implements the necessary management functions enabling policy initialization and update if necessary.

The overall functionality of SDU Protection is split into two operations:

- SDU Protection - For serialized PDU (sPDU or SDU), it computes a protected SDU (pSDU) that can be sent through the port of the underlying IPCP. It uses the SDU Protection policy associated with the SDU's N-1 flow to perform all the necessary operations on the serialized PDU.
- SDU Verification - For protected SDUs received from the underlying IPCP it computes the serialized PDU and provides it to RMT for further processing. If validation fails, it provides a reason and further diagnostic information.

SDU Protection workflows are simple. There is a workflow for each direction of processing. Figure 23, "SDU Protection Workflow Diagram" provides a visualization of both workflows.

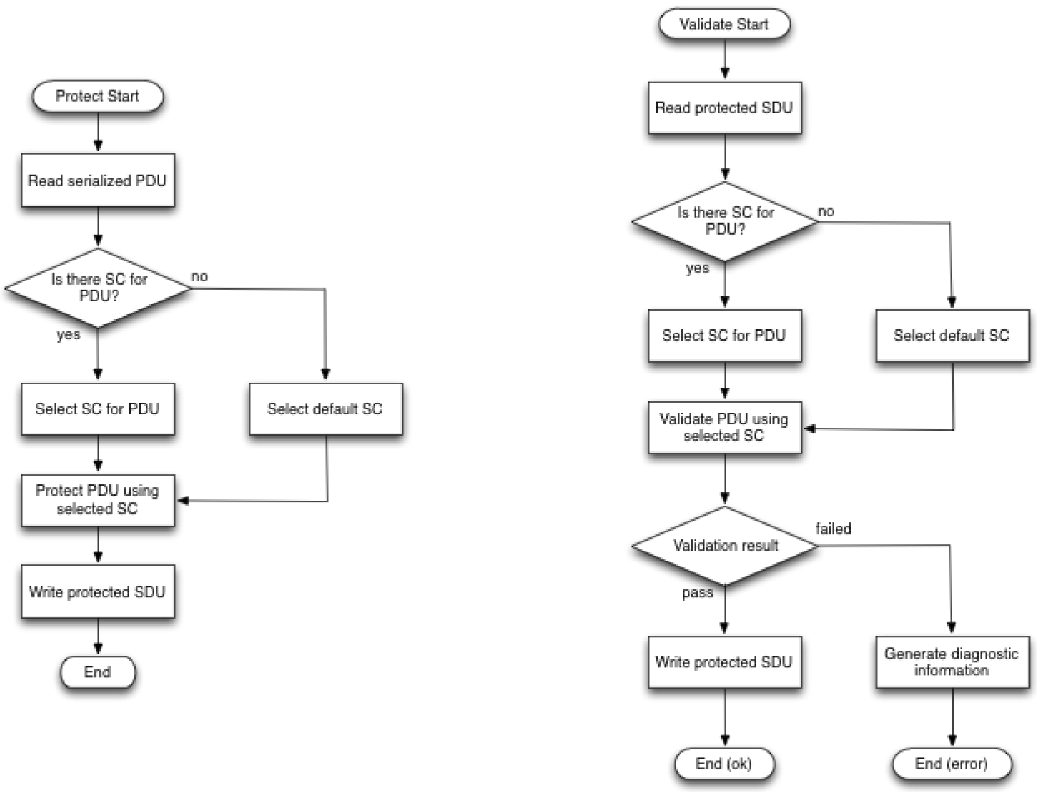

Figure 23. SDU Protection Workflow Diagram

- The SDU Protection workflow starts with a serialized PDU that is provided by the SerDes Module. To process the serialized PDU, SDU Protection has to find the Security Context associated with the PDU's flow. Applying SDU Protection is done according to the information provided by the Security Context. This contains information on the methods for TTL computation, content protection, data integrity computation, or compression and their parameters, such as encryption and integrity keys. If a Security Context is not found for the flow, then the default Security Context is used. This default Security Context provides TTL-based lifetime control and CRC calculation for data integrity computation.
- The SDU Verification starts to process new incoming (protected) SDUs. For this SDU, the Security Context needs to be retrieved in order to apply correct SDU validation function. If found, parameters and methods for validating protected SDU are taken from Security Context found by using the identified flow as a key. If a Security Context cannot be found then the default Security Context is used. Note that this may lead to an error if communicating parties have not properly synchronized their security contexts. Applying methods from the Security Context yields to a serialized PDU if SDU passes all validation steps. If some of the validation steps fail, then an error is reported and additional diagnostics information is provided.

5.2.2. SDU Protection Interfaces

The SDU Protection Container defines two interfaces, namely, `SduProtectionControl` and `SduProtectionData`. The first interface contains functions to modify the security settings of N-1 flows. The second interface is used to handle data to be protected or verified by the SDU protection module. Because SDU protection resides at the bottom of the IPCP, it can distinguish the SDUs using the outbound/inbound port. Thus all operations are related to a port object defined by means of the port id and N-1 DIF. The `SduPort` structure is defined as follows:

```
struct {
    uint32 dif_id;
    uint32 port_id;
} SduPort;
```

The `SduProtectionControl` interface provides a way of specifying which policy will be used with the `SduPort` and of setting up a newly instantiated policy with the necessary parameters. The interface is defined as follows:

```
enum { SDUPPS_ACTIVE, SDUPPS_KEY_MISSING, SDUPPS_LNONCE_MISSING,
 SDUPPS_RNONCE_MISSING } SduProtectionPolicyStatus;
interface {
    SduProtectionResult ResetSduPortProtection(in SduPort port_id)

    SduProtectionResult SetSduPortProtection(in SduPort port_id, in
 SduProtectionPolicy policy)

    SduProtectionResult GetSduPortProtection(in SduPort port_id, out
 SduProtectionPolicy policy, out SduProtectionPolicyStatus status)

    SduProtectionResult SetSduPolicyAttribute(in SduPort port_id, in
 string name, in byte[] value)

    SduProtectionResult GetSduPolicyAttribute(in SduPort port_id, in
 string name, out byte[] value)

    SduProtectionResult ApplySduPortProtection9in SduPort port_id)

} SduProtectionControl;
```

- **ResetSduPortProtection** removes all information associated with the port id. This function should be called when a flow is deallocated. After calling this function all information related to SDU Protection is removed and the SDU Protection module uses the default policy for all subsequent SDUs.

- **SetSduPortProtection** associates specified SDU protection policy settings to the specified port id. Setting an SDUProtectionPolicy creates a new instance of the policy, but this policy is not used until it is fully initialized.
- **GetSduPortProtection** gets information about the SDU Protection Policy associated with the specified port id.
- **SetSduPolicyAttribute** sets the Sdu Protection Policy attribute of the given name.
- **GetSduPolicyAttribute** gets the Sdu Protection Policy attribute of the given name.
- **ApplySduPortProtection** applies changes to settings of the SDU Protection Policy. This function serves for updating policy methods according to settings performed by `SetSduPolicyAttribute`.

The `SduProtectionData` interface is defined as follows:

```
interface {
    SduProtectionResult ProtectSDU(in SduPort port_id, in SduData in_sdu,
 out ProtectedSdu out_sdu);

    SduProtectionResult VerifySDU(in SduPort port_id, in ProtectedSdu
 in_sdu, out SduData out_sdu);

} SduProtectionData;
```

The meaning of `SduProtectionData` operations are as follows:

- **ProtectSDU** performs protective operations according to the SduPolicy assigned to the `SduPort` on input `SduData`. The result is provided in `ProtectedSdu`.
- **VerifySDU** verifies provided `ProtectedSdu` according to the SduPolicy instance associated with the `SduPort`.

5.2.3. Report of SDU Protection Operations: The Results and Error Codes

To report the result of SDU Protection operations and specify possible errors, the following enumeration is defined.

```
enum { SDUP_SUCCESS,
        SDUP_HMAC_VERIFICATON_ERROR,
        SDUP_DECRYPTION_ERROR,
        SDUP_COMPRESSION_ERROR,

        SDUP_FLOW_NOT_FOUND,
        SDUP_FLOW_EXISTS,

        SDUP_KEY_TOO_SHORT,
        SDUP_NO_ROOM,

        SDUP_ACCESS_DENIED,
        SDUP_OTHER_ERROR,
} SduProtectionResult
```

where

- **SDUP_SUCCESS** represents that no error occurred during SDU Protection operation
- **SDUP_HMAC_VERIFICATON_ERROR** represents the case when the message digest field and computed digest of the SDU differ. This can represent a situation when the SDU was modified in transit
- **SDUP_DECRYPTION_ERROR** stands for an error found during decryption of SDU protected data,
- **SDUP_COMPRESSION_ERROR** represents any error that occurred during decompression of SDU data. This may occur if different methods were used for compression and decompression of the data
- **SDUP_FLOW_NOT_FOUND** for operations specified for a flow. It means that the specified flow does not exist.
- **SDUP_FLOW_EXISTS** is used when the specified flow already exists. It cannot be create twice.
- **SDUP_KEY_TOO_SHORT** means that the provided key is too short.
- **SDUP_NO_ROOM** informs that SDU Protection module has not available resources to complete the requested operation.
- **SDUP_ACCESS_DENIED** means that the operation cannot be completed because access was denied.
- **SDUP_OTHER_ERROR** represents other errors that can occurs during verification of SDU.

5.3. SDU Protection Policies

SDU Protection performs operations as specified in the SDU protection policy set for the communication port. Two policies are defined.

5.3.1. Basic SDU Protection Policy: Simple CRC and TTL

Name: SDUP-CRC-TTL

Title: Simple CRC and TTL

Brief Description: This policy computes or checks the CRC on the SDU using the specified CRC polynomial. It also computes and checks TTL.

Domain of Applicability: This module might be used in a DIF with a lower layer subject to bursty errors and when no additional SDU protection is necessary. Therefore, only error checking and lifetime limiting will be provided by this policy. Because this policy does not require advanced configuration, it is often used as a default SDU protection policy.

Constraints and Assumptions: This module depends on the characteristics of well-chosen CRC polynomials. A CRC of n-bits is able to detect all 1 and 2 bit errors, all odd numbers of errors and all errors with a burst less than n bits in length, and will only fail to detect 1 in 2^n other patterns of errors. A CRC of n-bits should not be used with PDUs with length greater than 2^(n-1).

Policy Specifications: This policy computes CRC-16 and maintains TTL. Therefore it prepends two fields to any SDU.
CRC value is an n-bit unsigned integer representing the computed CRC value using the CRC-16-ANSI algorithm. This value is computed over SDU content including the TTL value. Thus, the TTL value should be determined first. The TTL value is an 8-bit unsigned value representing a number of hops remaining.

The structure of protected SDU is defined as:

```
struct {
    byte[CRC_LEN] crc;
    uint16 ttl;
    Pdu pdu;
} CrcTtlSdu
```

Management Elements

This module expose the following management elements that are used for setting the policy:

- string `PolynomialName` : a name of polynomial used for CRC calculation
- uint16 `ITTL` : an initial value of TTL

The module also contain common counters exposed through management elements:

- uint64 `SentSDUs` : total number of sent SDUs
- uint64 `SentOctects` : total number of sent octets
- uint64 `ReceivedSDUs` : total number of received SDUs
- uint64 `ReceivedOctets` : total number of received octets.
- uint64 `ReceivedErrors` : number of SDUs containing error

Outbound Specification:

When processing a new PDU from RMT's serialization module, this policy calculates a CRC for the PDU and adds a TTL value. Then the SDU is passed to the (N-1)-DIF through the specified destination port.

Inbound Specification:

When processing an incoming SDU, this policy first calculates the CRC and compares it with the values in the incoming SDU. Then the policy checks TTL. If both checks succeed then the content of the SDU is relayed to RMT's deserialization for further processing.

5.3.2. Cryptographic SDU Protection Policy: AES Counter Mode

Name: SDUP-CRYPTO-AES-CTR

Title: Cryptographic SDU Protection Policy based on AES Counter Mode

Brief description: This policy protects SDUs by using cryptographic algorithms to prevent eavesdropping and tampering. Because of the way the SDU Protection Policy processes data, only counter-mode is supported. In this policy the AES algorithm is provided in two lengths: either 128 or 256. This is similar to AES utilization in TLS [RFC3268]. For message integrity MD5 or SHA1 in different key lengths can be selected.

Domain of Applicability: This module might be used in a DIF with a lower layer that does not provide any security measures and when the security measures should be provided for the current DIF. Note that this kind of security represents IPCP to IPCP protection and not AE to AE protection. By applying this protection the size of SDU increases by 24—28 bytes (depending on the HMAC algorithm applied).

Constraints and Assumptions: This policy provides cryptographic algorithms to prevent eavesdropping and tampering. It can be configured with predefined combinations of encryption and integrity algorithms to provide the required security and computation costs.

AES-CTR has many properties that make it an attractive encryption algorithm for use in high-speed networking. AES-CTR uses the AES block cipher to create a stream cipher. Data is encrypted and decrypted by XORing it with the key stream produced by AES-encrypting sequential counter block values. AES-CTR is easy to implement, and AES-CTR can be pipelined and parallelized. AES-CTR also supports key stream pre-computation.

The security considerations for the use of AES-CTR are known from IPSec [RFC3686] and TLS/DTLS [modagugu]:

- Counter blocks must not be used more than once with a given key. This means that sequence number must not be used twice with the same key to encrypt different data.
- Pre-shared key is supported as encryption keys are generated from the master key, which itself is not used for encryption. Thus, because for each connection there are different pair of keys, counter blocks generated by client and server can safely overlap.
- Message integrity mechanisms must be employed because, as with other stream ciphers, data forgery is trivial without a message integrity mechanism.

The maximum number of SDUs that can be encrypted using the keys depends on the size of sequence number. As this value is set to 64-bits, it represents 2^64 SDUs. Once the sequence number is about to rollover, the Flow Allocator Instance managing the flow will create another EFCP connection with different cep-ids, preventing the rollover from happening. This operation is transparent to the SDU Protection module.

5.3.2.1. Specification:

This policy extends the SDU with new fields necessary for holding information related to cryptographic protection of the transmitted data.

The structure of protected SDU is defined as follows:

```
struct {
    uint64  seq_num;
    byte [HMAC_LENGTH] mac;
    byte [PDU_SIZE] payload;
} SduCryptoAesCtr;
```

`HMAC_LENGTH` is either 20 bytes for the SHA-1-based HMAC or 16 bytes for the MD5-based HMAC. The length of 'payload' corresponds to the PDU size, as using AES-CTR does not require padding.

Management Elements: This module exposes the following management elements that are used for setting the policy:

- string `CipherSpecification`: specifies which cipher suite to use. Possible values are AES-128-CTR, AES-256-CTR.
- string `MacSpecification`: message authentication code algorithms can be specified by selecting from one of the possible options: HMAC-MD5-128, HMAC-MD5-96, HMAC-SHA1-160, HMAC-SHA1-96
- string `MasterKey`: a string representing the Master key used for generating read and write keys for encryption as well as for HMAC computation.
- string `LocalNonce`: a local NONCE value used for generating keys
- string `RemoteNonce`: a remote NONCE value used for generating keys

The module also contain common counters exposed through management elements:

- uint64 `SentSDUs`: total number of sent SDUs
- uint64 `SentOctects`: total number of sent octets
- uint64 `ReceivedSDUs`: total number of received SDUs
- uint64 `ReceivedOctets`: total number of received octets.
- uint64 `ReceivedErrors`: number of SDUs containing error
- uint64 `SequenceNumberCounter`: a counter used as a source of sequence numbers for outgoing SDUs

Outbound Specification: When processing a new PDU from RMT's serialization module, this policy encrypts the content of the plain SDU and then computes the message integrity value of the encrypted SDU. Then the SDU is passed to the (N-1)-DIF through the specified destination port.

- **Encryption:** To encrypt a payload with AES-128-CTR, the encryptor sequentially partitions the plaintext (PT) into 128-bit blocks. The final PT block MAY be less than 128-bits. This partitioning is denoted as: `PT = PT[1] PT[2] … PT[n]`. In order to encrypt, each PT block is XORed with a block of the key stream to generate the ciphertext (CT). The keystream is generated via the AES encryption of each counter block value, with each encryption operation producing 128-bits of key stream. The encryption operation is performed as follows:

```
FOR i := 1 to n-1 DO
    CT[i] := PT[i] XOR AES(CtrBlk)
    CtrBlk := CtrBlk + 1
END
CT[n] := PT[n] XOR TRUNC(AES(CtrBlk))
```

The `AES()` function performs AES encryption with the fresh key. The `TRUNC()` function truncates the output of the AES encrypt operation to the same length as the final plaintext block, returning the leftmost bits.

The counter block (CtrBlk) is obtained as follows:

```
struct {
          uint48 local_nonce;  // low order 48-bits of LocalNonce string
          uint64 seq_num;
          uint16 blk_ctr;
       } CtrBlk;
```

- **Message Integrity Computation:** To compute message integrity, the selected HMAC method is use. The MAC is computed for payload only. HMAC is defined (according to RFC2104) as follows:

```
HMAC(K,m) = H(K XOR opad , H(K XOR ipad , m))
```

where

- `H` is a cryptographic hash function, e.g. SHA-1
- `K` is a secret key adjusted to block size of `H` (either padded or hashed), this key is obtained from the master key using key generation method described in Section 6.
- `m` is the Sdu payload to be authenticated

- `opad` and `ipad` are the outer and inner padding, respectively

Inbound Specification: When processing incoming SDU, this policy first calculates the CRC and compares it with the values in the incoming SDU. Then the policy checks the TTL. If both checks succeed then the content of SDU is relayed to RMT's deserialization for further processing.

- **Decryption:** Decryption is similar to encryption. The decryption of n ciphertext blocks is performed as follows:

```
FOR i := 1 to n-1 DO
    PT[i] := CT[i] XOR AES(CtrBlk)
    CtrBlk := CtrBlk + 1
END
PT[n] := CT[n] XOR TRUNC(AES(CtrBlk))
```

The `AES()` and `TRUNC()` operate identically as in the case of encryption. The counter block is obtained as follows:

```
struct {
    uint48 remote_nonce;  // low order 48-bits of RemoteNonce string
    uint64 seq_num;
    uint16 blk_ctr;
} CtrBlk;
```

- **Message Integrity Checking:** To check the message integrity, the checker first computes the integrity message using HMAC method defined in the Message Integrity Computation section and then it compares the result with provided value stored in `SduCryptoAesCtr.mac`.

5.3.3. Interdependencies with other components

The SDU Protection module requires that an SDU Protection Policy is selected for every flow and also that, in the case of a Crypto-based SDU Protection policy, all four methods are negotiated between the communicating parties and the master key and two nonces are agreed. This SDU Protection depends on the authentication component for obtaining the necessary information. It is the responsibility of the authentication module to provide the negotiated data. SDU Protection defines a control interface that can be used to set the SDU protection policy for each flow. This is described in the next section. MLS, described in Section 4, will define a new policy for SDU Protection.

5.3.4. Changes to the current IRATI stack for Integrating Other Policies

Because SDU Protection is entirely specified as a policy, the RINA specifications do not need to be modified. The IRATI stack currently has a hardcoded implementation of SDU Protection, which implements the Basic SDU Protection policy described in this document. This Basic SDU Protection policy is used as the default SDU Protection Policy in PRISTINE. Since the IRATI implementation is hardcoded, in order to allow the integration of other SDU Protection policies, a new mechanism enabling the execution of SDU Protection functions as defined in the SDU Protection Security Context needs to be implemented. Fortunately, since the SDU Protection functions are called from the Serialization/Deserialization module, modifications are limited to this module and SDU Protection is isolated from the rest of the IRATI stack.

5.4. Implementation of SDU Protection for PoC

The Proof of Concept implementation tests the feasibility of the use of the native Linux Crypto API for SDU encryption and integration of the basic SDU protection mechanism with the rest of the stack. Configuration of the implemented modules is part of the security manager configuration of the IPCM, which is also described in the Authentication part of this deliverable.

The following describes how to configure SDU Protection and the modifications made to enable us to conduct PoC tests.

5.4.1. Configuration of SDU Protection

As was just mentioned the configuration of SDU Protection is possible from the IPC Manager (IPCM) configuration file as part of the **securityManager** configuration dictionary, specifically using the **authSDUProtProfiles** dictionary. Here we can define the default profile as well as profiles to be used for specific N-1 DIFs. An example of the relevant (ignoring authentication configuration for clarity) configuration looks like this:

```
"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"
        }
    },
    "specific" : [
        {
            "underlyingDIF" : "110",
            "TTLPolicy" : {
                "name" : "default",
                "version" : "1",
                "parameters" : [ {
                    "name" : "initialValue",
                    "value" : "50"
                } ]
            },
            "ErrorCheckPolicy" : {
                "name" : "CRC32",
                "version" : "1"
            }
        }
    ]
}
```

The IPCM stores the parsed profiles in **AuthSDUProtectionProfile** objects that contain **PolicyConfig** objects for policies defined by SDU Protection:

```
class AuthSDUProtectionProfile {
public:
    std::string to_string();
    PolicyConfig authPolicy;

    PolicyConfig encryptPolicy;
    PolicyConfig crcPolicy;
    PolicyConfig ttlPolicy;
};
```

This configuration gets to the kernel through a Netlink message as part of a **DIFConfiguration** object. Finally in the kernel we store the profiles in the RMT instance using the **struct sdup_config** structure that points to the default profile and contains a list of the specific profiles. The individual profiles use the **struct dup_config_entry** structure:

```
struct dup_config_entry {
    // The N-1 dif_name this configuration applies to
    string_t *  n_1_dif_name;

    // If NULL TTL is disabled,
    // otherwise contains the TTL policy data
    struct policy * ttl_policy;
    u_int32_t   initial_ttl_value;

    // if NULL error_check is disabled,
    // otherwise contains the error check policy
    // data
    struct policy * error_check_policy;

    //Encryption-related fields
    struct policy * encryption_policy;
    bool        enable_encryption;
    bool        enable_decryption;
    string_t *  encryption_cipher;
    string_t *  message_digest;
    string_t *  compress_alg;
    struct buffer * key;
};
```

5.4.2. Extending the IPCP Structure

In order to be able to access the newly added configuration, the **struct ipcp_instance_ops** was extended with two new functions:

```
const struct name *          (* dif_name)(struct ipcp_instance_data * data);
int                          (* enable_encryption)(struct ipcp_instance_data
 * data,
                              bool                          enable_encryption,
                                     bool
 enable_decryption,
                                     struct buffer             * encrypt_key,
                                     port_id_t                   port_id);
```

Where the **dif_name** function returns the name of the DIF that the IPCP is part of. This is needed to identify which SDU Protection configuration should be used when using a specific N-1 DIF IPCP. This was implemented for all current IPCP instance types.

And the **enable_encryption** function was implemented only for Normal IPCPS and just calls the RMT function **rmt_enable_encryption** that will be described later. This message is exported to the user space components through the **RINA_C_IPCP_ENABLE_ENCRYPTION_REQUEST** Netlink message, and is used from the SecurityManager during Enrollment.

5.4.3. Modifications of RMT Structure

As previously mentioned, the SDU Protection profiles are stored in the RMT instance structure:

```
struct rmt {
        ...
        struct sdup_config *      sdup_conf;
        ...
};
```

This new data structure is managed by two new functions:

```
int                                 rmt_sdup_config_set(struct rmt *
 instance, struct sdup_config * sdup_conf)
static struct dup_config_entry * find_dup_config(struct sdup_config *
 sdup_conf, string_t * n_1_dif_name)
```

Where **rmt_sdup_config_set** is used to replace the currently used SDU Protection profiles with the newly provided ones. And the **find_dup_config** function finds a specific SDU Protection profile for the specified N-1 DIF.

Also previously mentioned is the **rmt_enable_encryption** function that manipulates the SDU Protection encryption policy associated with the specified N-1 port. Using this function we can enable and disable both encryption and decryption of SDUs separately, as well as change the encryption key.

The most significant change to the RMT implementation is in the creation of N-1 ports. The **struct rmt_n1_port** gained two new members:

```
struct rmt_n1_port {
        ...
        struct dup_config_entry * dup_config;
        struct crypto_blkcipher * blkcipher;
};
```

Where **dup_config** was explained earlier and **blkcipher** is a structure used by Linux Crypto API for data encryption. This is here only for the purpose of the PoC implementation and in the future both should be replaced with a single **SDU Protection Policy Data** structure.

To propely initialize the updated rmt_n1_port structure, the **n1_port_create** function now takes an additional parameter:

```
static struct rmt_n1_port * n1_port_create(port_id_t id, struct
  ipcp_instance * n1_ipcp, struct dup_config_entry * dup_config)
```

This parameter is directly stored in the `rmt_n1_port` structure and it is also used to initialize the `crypto_blkcipher` structure.

The new information stored in the `rmt_n1_port` structure is used in the `n1_port_write` and `rmt_receive` functions, where they are passed to the SerDes module as parameters.

5.4.4. Modifications to SerDes Module

The main part of SDU Protection mechanism is implemented in the SerDes module. This is to have the PoC mechanism in one place and the TTL and CRC mechanism were already present here.

First of all TTL and CRC mechanism are no longer **always on** or **always off** controlled by the kernel compilation; instead they use the configured SDU Protection profile. Both mechanism are disabled by default, and can be enabled by defining the **TTLPolicy** and **ErrorCheckPolicy** in the configuration profile. For now the ErrorCheckPolicy always assumes the use of the CRC32 mechanism. The TTLPolicy can be further configured by setting the **initialValue** parameter. The configured value is used as the initial TTL value when serializing PDUs.

No other modifications were made to the TTL and CRC mechanisms, they still use the same functions from the "du-protection.c" file and are still called after the PDU was serialized, adding additional data to the front of the serialized PDU. And analogously for deserialization.

The new mechanism added is SDU encryption. This mechanism is called after TTL and before CRC mechanisms. Same as for TTL and CRC it’s enabled if the **encryptPolicy** is defined in the configuration profile. For now the value of the **encryptAlg** parameter is ignored and AES128, in ECB mode is always used. It’s important to note here that this mode is not recommended for serious cryptographic work and was chosen just for the PoC implementation for it’s simplicity. Support for the CTR and other modes will be added later. The mechanism consists of two main parts:

- Size recalculation and padding. Since encryption interates over data in blocks of a set size, we need to pad our data to a multiple of this block size. For now we implement the PKCS#7 padding mechanism that appends N bytes of value N to the end of the message.
- Encryption (and its opposite) is implemented as a new function in the "du-protection.c" file and for now it simply encapsulates the function calls to the Linux Crypto API.

```
int dup_encrypt_data(const char             * src,
                     char                   * dst,
                     ssize_t                  src_size,
                     ssize_t                  dst_size,
                     struct crypto_blkcipher * blkcipher);

int dup_decrypt_data(const char             * src,
                     char                   * dst,
                     ssize_t                  src_size,
                     ssize_t                  dst_size,
                     struct crypto_blkcipher * blkcipher);
```

Logically the opposite operations happen in the reverse order during deserialization:

- ErrorCheck
- Decryption
- Padding removal
- TTL check

Implementation of Hashed Message Authentication Codes was skipped for the purpose of PoC since its functionality is similar to CRC. Continuing from the Proof of Concept, implementation be modified to define policy sets in line with the rest of the kernel stack. SDU Protection will then need to synchronize with Authentication and Enrollment. Some of this was already done (enrollment can enable/disable the encryption and set a new encryption key) but more work in this area is expected.

5.5. Next Steps for Cryptographic Activity: PoC Tests

The presented PoC implementation of SDU protection component consists of a container providing a suitable environment for attaching SDU protection functions. Implemented SDU protection component and a Crypto-based SDU Protection policy provide necessary functions to establish a secure channel between two peer IPCPs through the common underlying DIF.

Current PoC implementation aims to provide working SDU Protection component integrated with IRATI network stack. When implementing PoC, some simplifications were made. To complete implementation of Crypto-based SDU Protection Policy the Hashed Message Authentication method for ensuring data integrity will be implemented. Also, PoC implementation will be modified to define policy sets in line with the rest of the kernel stack. SDU Protection will then need to cooperate with Authentication and Enrollment components. Currently, SDU Protection is configured along with Authentication from the IPC Manager (IPCM) configuration file through authSDUProtProfiles, however some parameters of SDU Protection need to be negotiated during Enrollment and so more work will be done in this area. For cooperation with Authentication and Enrollment, SDU Protection specifies management interface. Functions of this interface provide the means of setting SDU Protection attributes as needed.

To test implemented SDU protection, basic Validation and Verification tests are proposed followed by Performance Evaluation Tests.

- Validation tests are focused on checking that SDU Protection PoC design comply with the requirements. Requirements for SDU Protection are specified in RINA documents as applying following functions to each SDU: i) lifetime limiting, ii) error checking, iii) data integrity protection, iv) data content protection.
- Verification tests prove that the SDU Protection component consistently operates without error according to its design specifications. Several unit tests will be created to check that individual functions of SDU Protection component are error-free. These unit tests will exercise functions by applying different arguments within the acceptable range as well as outside this range and check their results.

Besides applying outcomes the tests to the SDU Protection implementation, the PoC implementation will be adjusted to comply with the style of IRATI implementation. After finishing PoC tests and refining the source code of SDU Protection component, the implementation will be ready for the integration in IRATI distribution. This will enable the possibility to define and realize use cases in WP6 and perform integration tests.

6. Key Management

Two architectural options (Centralised and Distributed) were suggested in D4.1 for assuming the role of Key Server as the security sensitive entity. We further refined these options and will discuss these choices in the next version of this deliverable.

7. Resiliency and High Availability

This chapter details the work done on resiliency and high-availability in PRISTINE T4.3. It covers two main aspects: the resilient routing policy and the application of load balancing concepts to RINA.

Regarding resiliency, we decided to focus implementation efforts on the Loop-Free Alternate routing policy and omit the implementation of the Flow Liveness Detection policy. There are two reasons for this. Firstly, there is already a rudimentary liveness detection mechanism present in the IRATI implementation. While it is not implemented according to the structure proposed in D4.1 [D4.1] (in IRATI it is a function embedded in the Flow Allocator), its functions are still adequate to perform resilient routing. Secondly, the Flow Loopback Detection policy would also require some substantial changes to the Flow allocator, and will there be implemented as a part of the RINA traffic generator (rina-tgen) [rina-tgen] development in WP6, Task 6.2. This means that this function will be available at the DAF level, not the DIF policy level as originally intended. The work regarding resilient routing is described in Section 7.1, “Resilient Routing”

DAF Load Balancing was implemented for the main testing tool available in the PRISTINE repository, namely rina-echo-time. It will be further extended to a lightweight web server, NGINX in Task 4.3. The work regarding load balancing is described in Section 7.2, “Load Balancing”

7.1. Resilient Routing

7.1.1. IRATI Routing and Forwarding Tables

As a starting point, the IRATI prototype implements a rudimentary link-state routing policy based on the IS-IS protocol. Each IPCP maintains a graph representing its current knowledge of the connectivity of the DIF, which is updated by distributing Flow State Objects among IPCPs, which are kept in the Flow State Database (FSDB). Each vertex of the graph represents an IPC Process while each edge represents an N-1 flow between adjacent IPC Processes. Routes in the DIF are calculated by applying Dijkstra’s Shortest Path algorithm to the graph. These routes are used to fill the PDU Forwarding Table (PFT) with entries mapping an <address, QoS> pair to the list of N--1 ports that have to be used to reach the next hop in the path towards the destination. Every IPC Process computes its own PFT.

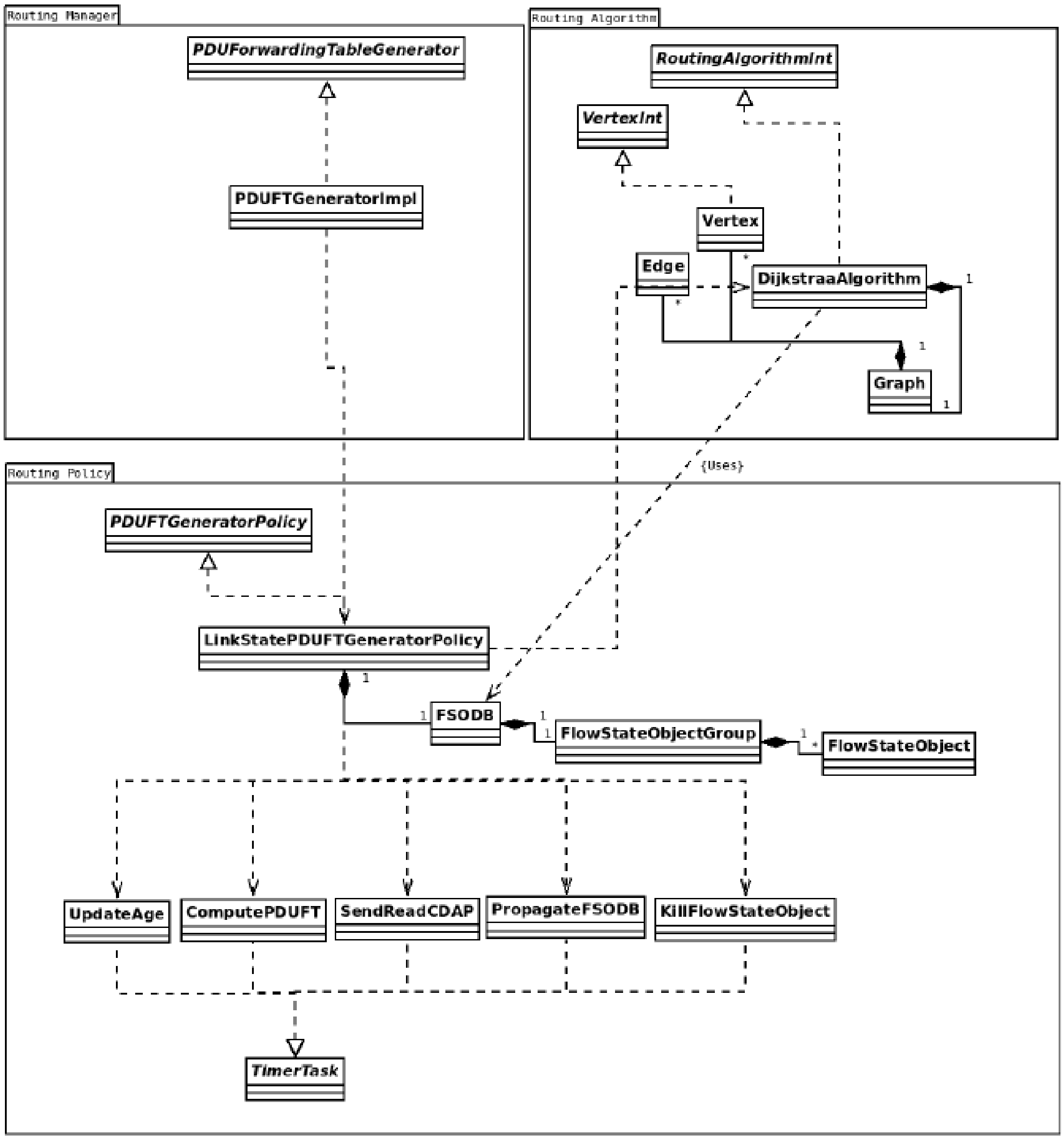

Figure 24. Organisation of the routing component in the IRATI prototype.

The organisation of the IRATI routing policy implementation is shown in Figure 24, “Organisation of the routing component in the IRATI prototype.”. The routing software follows a modular design that is partitioned in three components:

- The Routing Manager: responsible for the communication between the Routing Software module and the IPC Process which uses it.
- The Routing Policy: responsible for updating and maintaining the network graph. It sends / receives updated network connectivity information using the CDAP Protocol and changes the local representation graph when needed.
- The Routing Algorithm: responsible for computing the PFT from the network graph.

In the IRATI prototype, the routing table that is calculated from the FSDB consists of a list of routing table entries, where each routing table entry maps a destination address (for a certain QoS id) to a list of next-hop addresses. Multiple next-hops are possible per destination address for multicast support, but the available routing implementation does not use multicast routes, therefore the next-hops list of each routing table entry contains just one element, the unicast next-hop for a destination. The calculated routing table is passed to the Resource Allocator. Note that IRATI does not explicitly maintain a routing table, its entries are only used as an intermediary result between the FSDB and the PFT.

Starting from the routing table, the Resource Allocator computes the PDU forwarding table (PFT), by mapping each next-hop address to a port-id. This calculated PFT is modeled as a list of PDU forwarding table entries, where each entry maps a destination address and QoS id to a list of port-ids, very similar to what happens for the routing table. Multiple port-ids are possible per destination address to support sending the PDUs to multiple next-hops simultaneously (necessary for applications that use whatevercast communication).

The Routing component is an active component that performs the routing tasks based on timers and other asynchronous events (e.g. N-1 flow up/down). As an example, the default routing component starts by spawning different timer-driven tasks:

- A task to compute the routing table using a Shortest Path (SP) algorithm (Dijkstra algorithm has been chosen in the current implementation).
- A task to increment the age of the Flow State Objects (FSOs) received from the neighbor, in order to remove stale entries.
- A task to propagate the FSOs stored in the FSDB.

Detailed information on the IRATI routing policy can be found in IRATI deliverable D3.2 [IRATI-D32].

In order to support resilient routing, it is necessary to extend the current routing entry model so that each next-hop can be associated with one or more alternate next-hops (the Loop Free Alternates), to be used if the primary next-hop suddenly becomes unreachable - e.g. because of link failure, or neighbor node/IPCP crash. The current PDU forwarding table entry model also needs to be updated so that each port-id can be associated with one or more alternate port-ids, to be used if the flow represented by the primary port-id is unavailable.

7.1.2. PRISTINE SDK: Limitations and Proposed Solutions for Routing Policy

The current implementation of the IPC Process's core functionalities requires some modifications in order to fully support routing policies. Two obstacles have to be addressed:

1) Currently, setting up a new N-1 flow between two IPCPs is very intertwined with enrolling two IPCPs. There is no way to choose the connectivity graph for flows that will be used for layer management. 2) The RIB daemon does not support fine-grained control over the objects that are added to the RIB. For instance, FSOs have to be propagated at a certain interval, but there is currently no way to specify a propagation interval to the RIB daemon.

In order to overcome the first obstacle changes have to be performed to two main components: enrollment and the N-1 flow manager, which is part of the resource allocator implementation. Upon completion, enrollment currently sends all dynamic information, such as the FSDB, to the new member of the DIF. Enrollment will be modified to be marked as completed right before the sending of the dynamic information. Then according to policy, one or more N-1 flows will be setup to other IPC Processes in the DIF. This policy set will be implemented in the N-1 Flow Manager. We envision a few implementations for this policy set, to be able to investigate their advantages and disadvantages:

- Connect to all other IPCPs that share a common N-1 DIF
- Connect to a subset of the previous, with a fixed limit on the number of N-1 flows that need to be established
- Use a distance metric with the address as input to select the N-1 flows to setup
- Select the IPCPs to allocate an N-1 flow to in such a way that the graph is k-connected.

The second obstacle will be tackled by extending the RIB daemon API. It will allow specifying a policy set that manages subtrees of the RIB. In the case of the LFA routing policy, this will be managing the FSDB; the propagation of FSOs at a certain interval, the aging of FSOs, the removal of stale entries.

7.1.3. Loop Free Alternates Policy, the Updates

The original specification from D4.1 called for the Loop Free Alternates (LFA) policy to listen to the following events: N-1 flow allocated N-1 flow deallocated N-1 flow up

N-1 flow down Flow State Database has changed Upon revision, we removed the flow allocated and flow deallocated events to be accessed by a routing policy, in order to control assigning flows for data transfer. Upon flow allocation, the new flow will not automatically be announced to the routing policy. This allows to have explicit topology control for the forwarding of PDU’s in a DIF. The revised LFA policy will therefore only listen to the following events: N-1 flow up N-1 flow down Flow State Database has changed

7.1.4. Routing Software Specification and Implementation

7.1.4.1. User Space, Interfaces

In IRATI, the API between the IPC Process core and the routing plugin is minimal - the IRoutingPS abstract class. The key method exposed by this interface is set_dif_configuration(), that is invoked from the IPC Process core to start the Routing component. The API minimality reflects the fact that routing in RINA is all policy.

The introduction of a resiliency algorithm does not modify the interface defined by the IRoutingPS class, nor extend the overall interface between the IPC Process core (the fixed/common part) and the plugins (the policies). Instead, an interface internal to the Routing component - the IResiliencyAlgorithm abstract class - is added to abstract the operation of a resiliency algorithm, in addition to the already existing internal interface for the computation of the (initial) routing table.

The IResiliencyAlgorithm class exposes the extendRoutingTable method, which is used to insert additional next hops (e.g. loop-free alternates) to the routing table computed by the main routing algorithm.

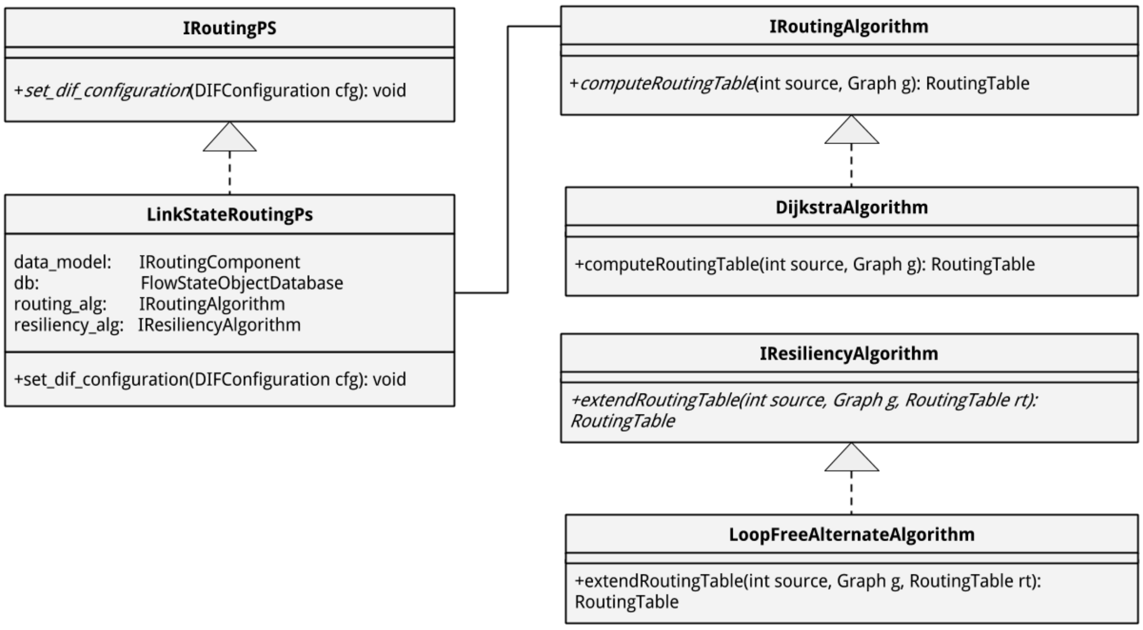
IRoutingPS
+set_dif_configuration(DIFConfiguration cfg): void
LinkStateRoutingPs
data_model: IRoutingComponent
db: FlowStateObjectDatabase
routing_alg: IRoutingAlgorithm
resiliency_alg: IResiliencyAlgorithm
+set_dif_configuration(DIFConfiguration cfg): void
IRoutingAlgorithm
+computeRoutingTable(int source, Graph g): RoutingTable
DijkstraAlgorithm
+computeRoutingTable(int source, Graph g): RoutingTable
IResiliencyAlgorithm
+extendRoutingTable(int source, Graph g, RoutingTable rt): RoutingTable
LoopFreeAlternateAlgorithm
+extendRoutingTable(int source, Graph g, RoutingTable rt): RoutingTable

Figure 25.

7.1.4.2. User/Kernel Interface, Data Structures

In the IRATI prototype (See Section 7.1.1, "IRATI Routing and Forwarding Tables"), the Resource Allocator (RA) is implemented in userspace, while the RMT is implemented in kernelspace. Upon receiving input from the Routing component (e.g. routing table), the RA generates the corresponding configuration for the PDU fowarding policy in the RMT component. This is implemented using a netlink message (currently referred to as MOD_PFTE), sent by the RA to to the kernel in order to configure RMT. The current data structures used to support routing and forwarding (in both kernelspace and userspace) are, however, tied to a specific implementation, reflecting the default routing policy and RMT policies. A PDUForwardingTableEntry userspace data structure is used to hold an entry of the default RMT PDU forwarding policy, which assumes destination-based routing/forwarding. A similar data structure exist in kernel space to directly implement RMT PDU processing. Consequently, the current format of the MOD_PFTE message also reflects the structure of the PDUForwardingTableEntry. However, PRISTINE research efforts in the routing and forwarding area envision different policies for Routing, Resource Allocator and PDU forwarding. This results in different requirements for the userspace and kernelspace data structures and the MOD_PFTE message.

First, we detail the format of the new MOD_PFTE message. The format of this message has to be flexible to support a wide range of possible routing policies, particularly the ones we envision in PRISTINE's scope. It should convey all the information necessary to configure any PDU forwarding policy, independently of the specific policy implementation. The way the MOD_PFTE message is interpreted in particular, is policy-implementation-specific.

The current format of the IRATI MOD_PFTE message is

```
struct mod_pdufte_entry {
   unsigned int destination_address;
   unsigned int qos_id;
   list<unsigned int> port_ids;
}

struct mod_pdufte {
    list<mod_pfte_entry> entries;
}
```

that is a list of PDU forwarding table entries.

For resilient routing, a format has been chosen to make it possible to support alternate port-ids:

```
struct alt_port_ids {
     list<unsigned int> alternatives; /* First entry is the primary one */
}

struct mod_pfte_entry {
   unsigned int destination_address;
   unsigned int qos_id;
   list<alt_port_ids> port_ids;
}
```

The port-ids contained in struct alt_port_ids are intended to be the different alternatives, sorted in failover order.

Apart from T4.3, interaction with WP3 identified the following PRISTINE tasks that will make direct use of this message in their research effort:

- T3.2 Multipath routing
- T3.3 Topological Addressing

For T3.2 multipath routing, the current format for struct mod_pdufte_entry is sufficient, since the list of port-ids can be used to support the multiple paths.

For the purpose of T3.3 topological addressing research, multiple formats have been proposed.

For topological addressing

```
struct mod_pfte_entry {
    unsigned int neighbor_address;
    unsigned int port_id;
}
```

to support forwarding not based on destination address, but rather on topological distance information.

For circuit-based switching:

```
struct mod_pfte_entry {
    unsigned int circuit_id;
    list<unsigned int> port_ids;
}
```

where a circuit identifier is used in place of a destination address.

7.1.4.3. Kernel Space Software Structure

The current prototype provides a basic PDU forwarding table implementation, based on a list of entries, where each entries contains a list of port-ids. In order to support resilient routing, accordingly with what specified in the previous sections, the entry data structure has to be extended so that each primary port-id (more than one port-ids are present in case of multicast) in the list can have one or more alternate port-ids.

Currently, the policy set only contains the following behavioural policies (hooks):

```
int (* next_hop)(struct pft_ps * ps,
                 struct pci *    pci,
                 port_id_t **    ports,
                 size_t *        count);

/* Reference used to access the PFT data model. */
struct pft * dm;
```

and uses the dm to access the hard-coded PFT implementation contained in the pft.c file. The PFT is implemented as a list of entries, where each entry maps a destination address to a list of next hops. However, the PFT implementation really depends on the kind of forwarding table being used - a resilient forwarding table (to be used with LFA) needs each entry to contain either a primary port-id and an alternate port-id. For this reason, the policy set interface was extended to make it possible to keep the table in its internal implementation - and consequently not hard-coded into the stack. In order for this to be possible, it was also necessary to add further hooks in the policy set to support update to the PFT internal implementation.

Note: a performance software implementation would make use of hashtables. Note: a more robust implementation would (logically) separate pdu forwarding tables (and ideally all data structures) per qos-id to minimise interactions of one qos-id with another.

7.1.5. Initial PoC Evaluation of the LFA Policy

In order to explore the feasibility of the LFA policy in the context of the routing implementation provided by the IRATI stack, an initial implementation of the LFA core algorithm has been developed. It is scheduled to be integrated in the pristine-1.3 public release.

In the following, the IPC process on which the routing and LFA computation happens will be referred to as source node, while the term neighbor of a node will refer to another node towards which the first node has a direct link (N-1 flow) in the DIF graph.

Finding LFA nodes requires the computation of the distance vector rooted in the source node and and the distance vectors rooted at each of source node's neighbors. A distance vector rooted at node X maps each node Y in the DIF graph to the minimum distance between X and Y.

The original Dijkstra implementation is structured in the following steps: Computation of the distance vector (with predecessor information) for the specified root node Use the predecessor information computed in step 1 to compute the next hop for the root node towards all the other nodes

In the IRATI implementation, however, the two steps were tightly coupled, so it was not possible to obtain the distance vector without computing the next-hops. For this reason, some initial refactoring for the original implementation has been carried out to allow faster computation of distance vectors (skipping next-hop computation, which is not needed for LFA).

The following pseudocode outlines the implementation of LFA core algorithm - e.g. the computation of LFA nodes for the source (local) node:

```
src_dist_vec ← computeDistVec(graph, src_node)

foreach neigh in neighbors(src_node) {
    neigh_dist_vecs[neigh] ← computeDistVec(graph, neigh)
}

foreach node not in neighbors(src_node) {
    foreach neigh in neighbors(src_node) {
        if neigh_dist_vecs[neigh][node] < src_dist_vec[neigh] +
 src_dist_vec[node] and neigh not in nexthops[node] {
            add neigh to LFA node towards node
        }
    }
}
```

As the pseudocode reports, the algorithm is organized in two steps: Compute the distance vector rooted at the source node and and the distance vector rooted at each of the source node's neighbors. This step requires as input the identifier of the source node and the DIF graph. For each remote node (i.e. a node that is not a neighbor of the source node, and this can be reachable over LFA nodes), try to see if some source node's neighbor - excluded the one that is already the next-hop towards the remote node - satisfies the LFA inequality. If the condition holds, the neighbor is added as LFA node for the remote node. This step requires as input the distance vectors computed at step 1 and the original routing table computed by the routing component (which contains the next-hops towards each node).

The IRATI build infrastructure already provides a unit test infrastructure for the routing algorithm, so that there is no need to setup a real scenario - with virtual or physical machines running the stack - to verify the functionality of the routing algorithms. The unit tests can be carried out by means of the make check commands of the rinad software package.

Therefore, the already existing unit tests have been extended to also check the correct functionality of the LFA algorithm.

The following test graph has been used for the LFA unit test, where the source node is identified by "1", and all the links have equal cost (1):

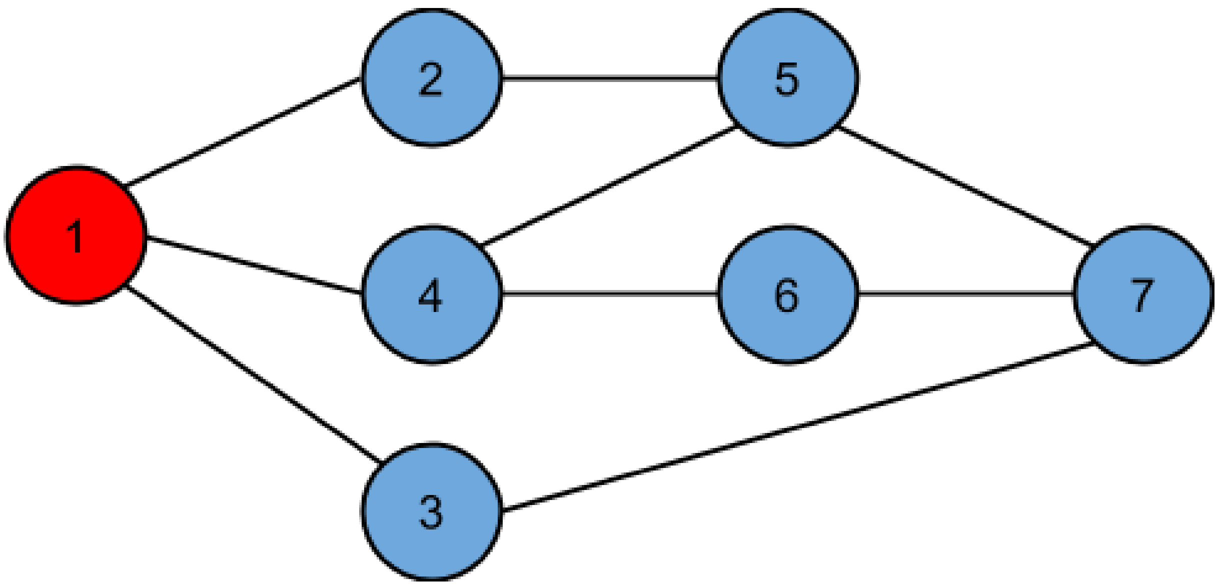

Figure 26. Test topology for LFA algorithm

The make check command produces the following output (only the part relevant to the test case described above is reported)

```
[...]
Dest: 2, Cost: 1, NextHops: [2, ]
Dest: 4, Cost: 1, NextHops: [4, ]
Dest: 3, Cost: 1, NextHops: [3, ]
Dest: 5, Cost: 2, NextHops: [2, ]
Dest: 6, Cost: 2, NextHops: [4, ]
Dest: 7, Cost: 2, NextHops: [3, ]
22984(1432291094)#ipcp (DBG): Node 3 selected as LFA node towards the
 destination node 5
22984(1432291094)#ipcp (DBG): Node 4 selected as LFA node towards the
 destination node 5
22984(1432291094)#ipcp (DBG): Node 3 selected as LFA node towards the
 destination node 6
22984(1432291094)#ipcp (DBG): Node 2 selected as LFA node towards the
 destination node 7
22984(1432291094)#ipcp (DBG): Node 4 selected as LFA node towards the
 destination node 7
22984(1432291094)#ipcp[1].lsr-tests (INFO):
 getPDUTForwardingTable_MoreGraphEntriesLFA_True test passed
[...]
```

The first part of the output shows the routing table (next-hops) as normally computed by the routing component. The second part reports the results of the LFA algorithm. In this case: Neighbors 3 and 4 have been selected as LFA nodes for remote node 5. Neighbor 3 as has been selected as LFA for remote node 6. Note that neighbor 2 (which is not the next-hop for remote node 6) does not satisfy the LFA condition. Neighbors 2 and 4 have been selected as LFA for remote node 7.

7.2. Load Balancing

In order for balancing the load between servers in a data centre scenario, currently an additional entity/node is being used which is called Load Balancer (LBR). The LBR has one or more public routable IP addresses and has one or more servers behind it. The limitation behind this model is that the servers and LBR need to be in the same layer 2 domain. If one or more servers are not in the same layer 2 domain, then such servers would not be able to see the addresses of clients they should be connected to. Therefore, in order for LBR to connect with a server in other layer 2 domains, the packets have to pass through a layer 3 node/router. RINA architecture does not have this limitation. In RINA, servers can be placed anywhere. Application names are location and layer independent; therefore servers can always see the client applications.

7.2.1. DAF-Based Load Balancing

Introducing additional standalone nodes such as LBR in the end-to-end path might create some performance degradation specifically towards the delay and loss experienced by traffic flows, possibly due to excessive processing and load at the LBR. Moreover, in order to avoid a single point of failure and to further balance the load, redundant/additional LBRs are normally deployed in the data centres. This might make the LB a more costly solution and can be difficult to maintain. Unlike in current architectures, load balancing in RINA based data centres is envisaged to be implemented at the DAF level, rather than by deploying additional node/s. DAF based load balancing will utilise a distributed application facility operating at various nodes on the network, which will coordinate with the resources and can redirect network traffic towards lightly loaded servers to make efficient use of resources.

7.2.2. Implementation of DAF-Based Load Balancing

Here, load balancing is defined as the process of workload distribution across multiple available resources/servers. It tries to maximise resource scalability and availability, and makes more efficient use of resources. The LBR distributes load/traffic among more than one available instances of the same server. We envisaged that load balancing can be deployed in a DAF in RINA. As a proof of concept, we initially conducted an experiment using two instances of rina-Echo-Time server running on two distinct virtual machines and one instance of rina-Echo-Time client running on a third virtual machine. In this experiment, the LB-DAF is not implemented; however, a similar functionality was implemented in the rina-echo-time client application. In this experiment, if a user on the client side wants to exchange 1000 packets with the server, the load balancing function initiates two threads and exchanges 500 packets with each server. We explain below how this experiment was conducted.

There are no changes made to the rina-echo-time application's server side implementation. On the client VM, the client side implementation of the application code is modified to initiate two distinct flows with each server instance. The client application process started two independent threads.

```
pthread_create ( &thread1, NULL, run_client, (void *) &arguments1)
pthread_create ( &thread2, NULL, run_client, (void *) &arguments2)
```

Here, arguments1 and arguments2 are pointers to a structure holding all the runtime arguments taken while executing the client application.

```
struct arguments {
string t_type;  // test type (perf, ping)
string s_apn;  // application process name for server
string c_apn;  // application process name for client
string s_api; // application process instance for client
string c_api;  // application process instance for server
string d_name;  // The name of the DIF to register at
bool reg;  // Register the application
boot qt;  // Suppress some output
unsigned int cnt;  // total number of packets to send
unsigned int sz; // size of packets to send
unsigned wt;  // time to wait between packets;
int gp; //  Gap of the retransmission window
int d_time;  // Deallocate the flow after specified time
};
```

The simple command to run the client is as follows:

```
#./rina-echo-time -c 200 --server1-api 1 --client1-api 1 --server2-api 2
 --client2-api 2
```

It can also be given if we want client application instance 1 to be connected to server application instance 2:

```
#./rina-echo-time -c 200 --server1-api 1 --client1-api 2 --server2-api 2
 --client2-api 1
```

Each thread initiated a flow with one server instance and started sending and receiving echo messages. The *run_client* function was used to create an object of the Client class and call its constructor and *run* function.

```
void *run_client (void *parameters)
{
  struct arguments *args;
  args = (struct arguments *) parameters;
  Client c(args->t_type, args->d_name, args->c_apn, args->c_api,  args-
>s_apn, args->s_api, args->qt, args->cnt, args->reg, args->sz, args->wt,
 args->gp, args->d_time);

    c.run( );
    pthread_exit(NULL);
    return NULL;
}
```

We also setup three virtual machines over a virtual LAN. These machines are named as server1, server2 and client. Each application is enrolled with the same DIF named 'normal.DIF'. Application instance 1 for Echo Server started on server 1 and application instance 2 for Echo Server started on server 2 VM. IPC processes named 'test1.IRATI', 'test2.IRATI' and 'test3.IRATI' were created on server 1, server 2 and client VMs respectively. Each application instance is also registered at the respective IPC process. All this is done in the *ipcManager.conf* file as follows:

```
“applicationToDIFMappings”: [ {
“encodedAppName” : “rina.utils.apps.echo.server-1--”,
“difName” : “normal.DIF” }, …….. 

“ipcProcessesToCreate” : [ {
……..
“type” : “normal-ipc”,
“apName” : “test1.IRATI”,
“apInstance” : “1”,
“difName” : “normal.DIF”,
“difToRegisterAt” : [“100”]
} ………
```

After that, each IPC process is enrolled at 'normal.DIF'. This setup that is composed of three VMs is shown in Figure 27, “Load Balancing Evaluation Experiment”.

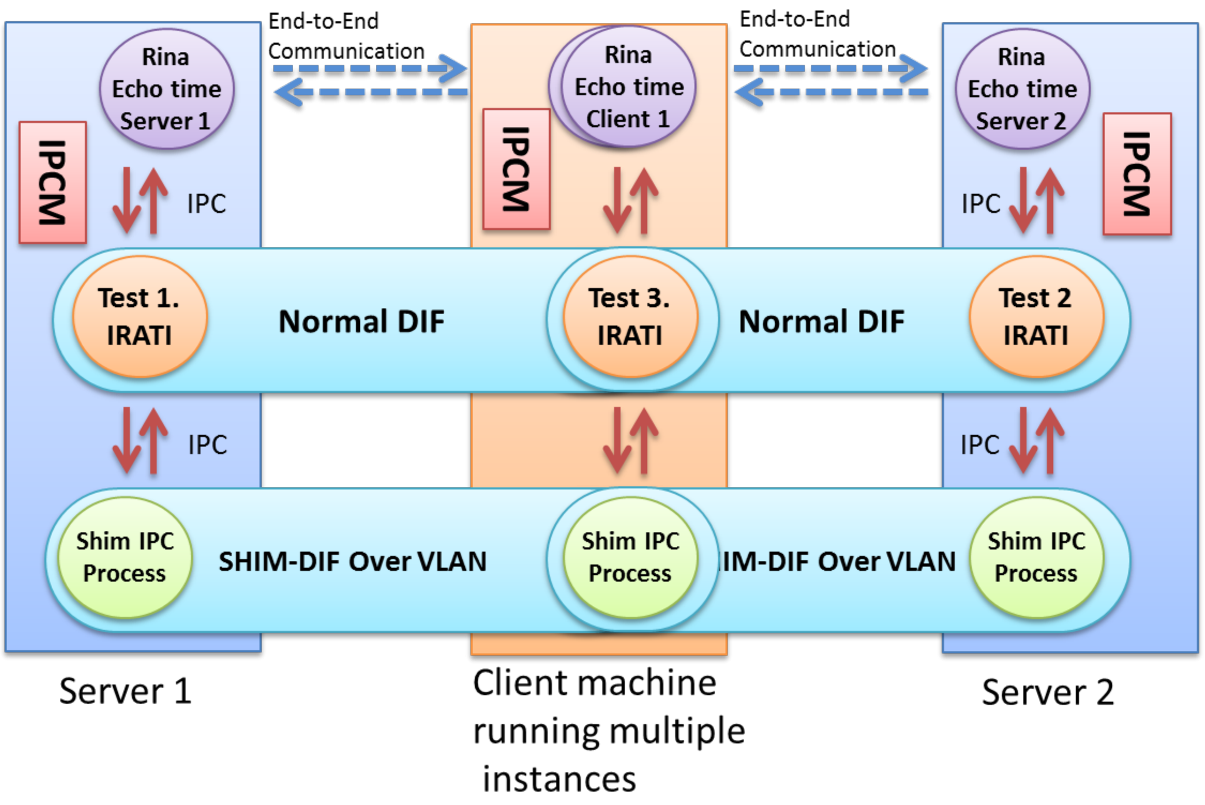

Figure 27. Load Balancing Evaluation Experiment

In this experiment, the connection initiation and load balancing have been carried out at the Application Process (AP) level. So the AP must be aware of the process names and instances of the servers in this case. The client AP requests for the flow allocation to each server application instance. In this request (as per current librina API) the AP needs to specify the *app_name*, *app_instance*, *server_name*, *server_instance*, *DIF_name*, and *QoS_spec*. Each flow to the server is distinct and independent as can be seen from the sequence numbers of packets for each flow in the log. In this way, it is the job of the AP to put the received packets in order.

If we transfer the responsibility of the load balancing task to the DIF, then the DIF must be aware of the number of instances of the servers and their locations. However, in the current implementation of librina, the AP needs to specify the server instance.

7.3. Next Steps for High Availability and Load Balancing Activities

7.3.1. High availability

In order to move towards high availability (HA) of IPC processes and DIFs in a RINA deployment, we performed an investigation into HA techniques used in GNU/Linux. More specifically, we looked into Corosync and Heartbeat. After some investigation, we found that these solutions do not translate to the recursive nature of RINA. The idea of deploying an IPCP in a virtualised environment and then cloning this to different systems broke down when trying to figure out how to do an implementation. The conclusion is that in RINA, high-availability would be more naturally implemented by using namespace resolution to anycast names. RINA envisioned namespace resolution from the onset, where a name can either resolve to a single AP (unicast), a set of AP's (broadcast), a member of a set (anycast) or a subset of a set. The overall name is therefore coined a 'whatevercast' name.

The current specification of whatevercast and multipoint flows is not very detailed. The objective of the work in the final period of PRISTINE is therefore to get a full specification of whatevercast, and a basic PoC implementation demonstrating the benefits for resiliency (IPCPs in whatevercast groups).

7.3.2. Load Balancing

We will port NGINX web server and Chromium browser with librina in order to make both of these work on RINA based systems. On the client side, we will implement a DAF with a Chromium browser for load balancing and bandwidth aggregation exercises. On the server side, we will use a NGINX web server in the same DAF. Please see Figure 28, "DAF-Based Load Balancing Scenario"

There are two aspects to consider for load balancing in RINA:

1. Re-ordering of received packets if a client connects to multiple servers and duplicated data packets from the servers are received by the client. This is the case when there are multiple servers for the same service under a single administrative domain e.g. www.google.ie and www.google.pk etc. The client application process can choose the server/s to connect to. For example, if there are two file servers having a specific file of size 2GB. The client may connect to both the servers and request half of the file from server 1 and the other half from server 2. DAF-based LB is application-specific load balancing and should be implemented in the client

application too. Using this approach should reduce the load on servers, enhance the throughput and aggregate the bandwidth if the flows adopt distinct paths. Because, if the flows pass through a common intermediate node, then the available capabilities at that node need to be shared among each flow that might cause performance degradation.

2. Selection of server instance to connect to if multiple clients contend for the same server. If a client does not give any server preference, e.g. it just wanted to access google.com, then the DAF Manager should decide which server instance to connect to and allocate a flow. By using this approach the DAF Manager has a better view of allocations and could balance the load at servers and eventually clients can experience better throughput.

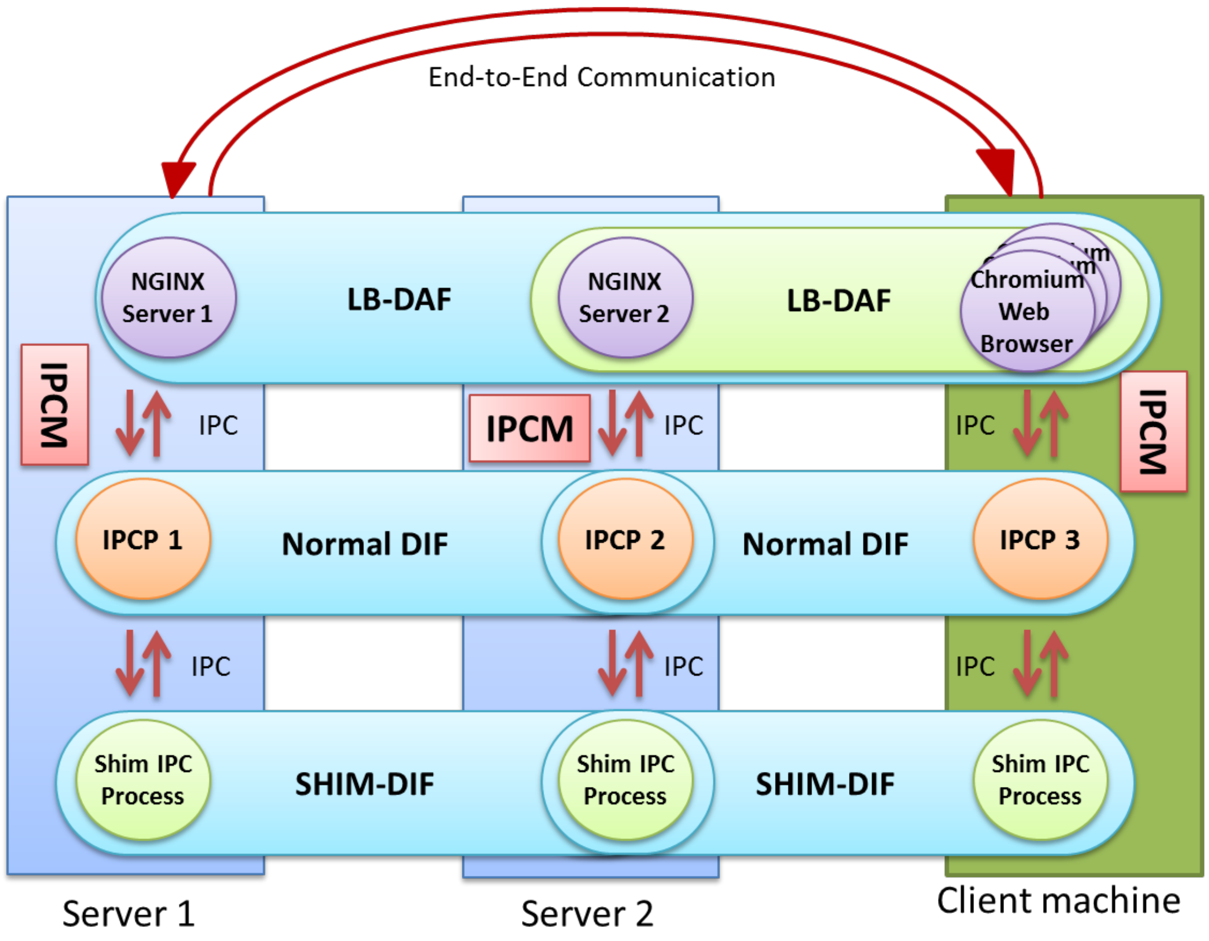

Figure 28. DAF-Based Load Balancing Scenario

The LB needs to be implemented at two steps, i.e., DAF and DIF levels. A DAF client AP chooses which server instance it needs to connect to. The client may choose more than one server to connect to in order to aggregate the bandwidth and balance the load. The DAF is more tightly coupled with the AP therefore putting out-of-order packets in the correct order can be done more effectively here. The DIF has to handle a lot more flows than the DAF, therefore it might become a bottleneck if it has to put the packets arriving from multiple paths in order for a single AP. Moreover, packets may have to wait longer in queues at the DIF while waiting for the packets delivered earlier than these packets.

The DIF is aware of the resources and number of instances of servers, therefore flow allocation and resource reservation needs to be done over here.

Load balancing in RINA should enable applications to connect to the most lightly loaded server. In order to do that, each instance of the server application must share its load statistics with the DIF it is enrolled with. Then the LB DAF can decide which server instance to connect to according to its load statistics.

8. Summary and Conclusions

The security requirements are analysed in T2.1 and reported in D2.1 [D2.1]. The PRISTINE reference framework was analysed in T2.2 and the results reported in D2.2 [D2.2] that included some of the security functions. D4.1 built on D2.2 and described the concepts and high-level design of security functions, mechanisms, and techniques. D4.2 provides further developments of these functions to meet the requirements enabling more secure and reliable networks. Below summarises the work carried out and reported in this deliverable related to these security functions mechanisms, and techniques. The future works are also sated.

Authentication: This is defined as the process of verifying the identity of IPC Processes that want to join a DIF. Six different authentication policies were proposed in D4.1. Among them, three authentication policies namely: no authentication required, authentication using asymmetric key, and authentication using password were specified, developed, tested and reported in this deliverable. Further work, such as developing other authentication policies inspired by the TLS handshake protocol and the iterations of experimental activities, will be conducted in WP4 and WP6.

Capability Based Access Control: Three scenarios for the use of CBAC have been specified in this deliverable. The scenario, when an AP needs to access other AP's resources in the same DAF, has been specified and implemented. Further work needs to be conducted for the verification tests of this scenario and specification and implementation of the other two scenarios namely: when an IPC Process requests to join a DIF and when an IPCP execute remote operations on the objects of a peer's IPCP RIB.

Multi-Level Security: D4.1 reported a number of MLS architectures that enable secure data sharing to be achieved on the common RINA infrastructure. There are two components that are needed to create these MLS architectures: Communications security and Boundary Protection Components (BPC). Design and specification of these two components are reported in this deliverable. Implementation is under way and the component tests will be conducted soon. The specification and implementation of communication security is believed to be straight forward given the RINA architecture. But regarding the BPC, enabling controlled sharing of data between classification levels in a DIF is more difficult. It requires coordinated policies in several RINA components. Deep content inspection is best performed at the application layer, i.e. the DAF layer. However, it is not currently possible to do this in a way that is transparent to applications, i.e. where the sending application does not sends its data directly to the BPC.

SDU Protection: The SDU Protection module is a part of the IPCP data path and protection is applied prior to exchange of data between two IPCPs of different DIFs. In this deliverable, a description of concepts, methods and algorithms used in the design, specification and implementation of the SDU protection module have been given. The software architecture, interfaces, and policies relevant to this component have been described. Two SDU protection policies are defined: Basic policy (simple CRC and TTL) and Cryptographic policy (AES Counter Mode). Both policies have been specified and implemented. The deliverable also reports on the plan for PoC tests.

Resiliency and High Availability: Two relevant aspects, namely resilient routing focusing on Loop-Free Alternate routing policy and load balancing focusing on DAF-Based Load Balancing, have been covered in this deliverable. The LFA-based policy has been specified, implemented and tested. High-availability of IPCPs and DIFs have also been investigated and realised. Further work on extending the scope of high-availability in terms of name resolution from anycast to whatevercast is envisaged. It is argued that DAF-based Load Balancing is best suited to RINA. An initial implementation and PoC evaluation have been conducted. Further tests are planned.

In summary, we will advance further towards the implementations and experimentations of security components, especially on the subjects identified above, conduct the foreseen in-house tests, and provide the modular security components to WP6.

References

- [D2.1] Diego Lopez, Editor. PRISTINE Consortium. Deliverable 2.1. Use Cases and Requirements Analysis. May 2014. Available online: http://ict-pristine.eu/wp-content/uploads/2013/12/pristine_d21-usecases-and-requirements_draft.pdf, accessed June 2015.
- [D2.2] Eduard Grasa, Editor, PRISTINE Consortium. Deliverable 2.2. PRISTINE Reference Framework. June 2014. Available online: http://ict-pristine.eu/wp-content/uploads/2013/12/pristine_d22-ref-framework_draft.pdf, accessed June 2015.
- [D2.3] Francesco Salvestrini, Editor. PRISTINE Consortium. Deliverable 2.3. Proof of Concept of the Software Development Kit. January 2015. Available online: http://ict-pristine.eu/wp-content/uploads/2013/12/pristine-d23.pdf, accessed June 2015.
- [D4.1] Hamid Asgari, Editor. PRISTINE Consortium. Deliverable 4.1. Draft Conceptual and High-Level Engineering Design of Innovative Security and Reliability Enablers. September 2014. Available online: http://ict-pristine.eu/wp-content/uploads/2013/12/pristine_d41-security-and-reliability-enablers_draft.pdf, accessed June 2015.
- [D5.1] Sven van der Meer, Editor. PRISTINE Consortium. Deliverable D5.1. Draft specification of common elements of the management framework. June 2014. Available online: http://ict-pristine.eu/wp-content/uploads/2013/12/pristine_d51-common-management-elements_draft.pdf, accessed June 2015.
- [D6.1] Miguel Angel Puente, Editor. PRISTINE Consortium. Deliverable D6.1. First iteration trials plan for System-level integration and validation. March 2015. Available online: http://ict-pristine.eu/wp-content/uploads/2013/12/pristine_d61_first_iteration_trials_plan_v1_0-1.pdf, accessed June 2015
- [DeepSec] Deep Secure XML Guard Brochure, http://www.deep-secure.com/wp-content/uploads/2014/06/xml-guard-brochure1.pdf, accessed June 2015.
- [DH] Diffie, W.; Hellman, M. (1976). "New directions in cryptography" (PDF). IEEE Transactions on Information Theory 22 (6): pp. 644–654. Available online: http://ee.stanford.edu/~hellman/publications/24.pdf, accessed June 2015.
- [Gollmann] D. Gollmann, “Computer Security”, Second Edition, John Wiley & Sons, November 2005.
- [IRATI-D32] Francesco Salvestrini, Editor. IRATI Consortium. Deliverable 3.2. Second Phase Integrated RINA Prototype over Ethernet for a UNIX-like OS. August

2014. Available online: http://irati.eu/wp-content/uploads/2012/07/IRATI-D3.2-v1.0.pdf, accessed June 2015.

- [LinkDD] “Interactive Link Data Diode – Connectivity Without Compromise”, Datasheet. Available online: http://www.baesystems.com/download/BAES_156410/interactive-link-brochure, accessed June 2015.
- [MAGEN] “MAGEN – the big cover-up: Masking technology developed in the Haifa Research Lab protects confidential data from unauthorized people.” Available online: http://www.research.ibm.com/haifa/info/200904_MAGEN.shtml, accessed June 2015.
- [Mansor] S. Mansor, et al., “Analysis of Natural Language Steganography”, International Journal of Computer Science and Security (IJCSS), Vol. 3: Issue 2, pp. 113-125, 2009.
- [modagugu] N. Modadugu, E. Rescorla. AES Counter Mode Cipher Suites for TLS and DTLS. June 2006. Internet-Draft. Available online: https://tools.ietf.org/html/draft-ietf-tls-ctr-01, accessed June 2015.
- [netlink] Linux Programmer’s Manual, http://man7.org/linux/man-pages/man7/netlink.7.html, accessed June 2015.
- [Nexor] Nexor Watchman Datasheet, http://nexor.co.uk/sites/default/files/Nexor%20Datasheet%20-%20Nexor%20Watchman%207.0.pdf, accessed June 2015.
- [ngenc] Next generation encryption. Cisco Systems. April 2014. Available online: http://www.cisco.com/web/about/security/intelligence/nextgen_crypto.html, accessed June 2015.
- [rina-tgen] RINA traffic generator. Available online: http://github.com/irati/traffic-generator, accessed June 2015.
- [openssl] OpenSSL libcrypto API. Available online: https://wiki.openssl.org/index.php/Libcrypto_API, accessed June 2015.
- [RFC1321] R. Rivest, "The MD5 Message-Digest Algorithm", RFC 1321, IETF, April 1992. Available online: https://www.ietf.org/rfc/rfc1321.txt, accessed June 2015.
- [RFC2401] S. Kent, "Security Architecture for the Internet Protocol", RFC 2401, IETF, November 1998. Available online: https://www.ietf.org/rfc/rfc2401.txt, accessed June 2015.
- [RFC3268] P. Chown, "Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)", RFC 3268, IETF, June 2002. Available online: http://tools.ietf.org/rfc/rfc3268.txt, accessed June 2015.

- [RFC3686] R. Housley, "Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP)", RFC 3686, IETF, January 2004. Available online: https://www.ietf.org/rfc/rfc3686.txt, accessed June 2015.
- [RFC4252] T. Ylonen, C. Lonvick, "The Secure Shell Authentication protocol", RFC 4252, IETF, January 2006. Available online: http://tools.ietf.org/rfc/rfc4252.txt, accessed June 2015.
- [RFC4253] T. Ylonen, C. Lonvick, "The Secure Shell Transport Layer protocol", RFC 4253, IETF, January 2006. Available online: http://tools.ietf.org/rfc/rfc4253.txt, accessed June 2015.
- [RFC4301] S. Kent, K. Seo, “Security Architecture for the Internet Protocol”, RFC 4301, IETF, December 2005. Available online: http://tools.ietf.org/rfc/rfc4301.txt, accessed June 2015.
- [RFC4306] C. Kaufman, "Internet Key Exchange (IKEv2) Protocol", RFC 4306, IETF, December 2005. Available online: https://www.ietf.org/rfc/rfc4306.txt, accessed June 2015.
- [RFC5246] T. Dierks, E. Rescola, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, IETF, August 2008. Available online: https://www.ietf.org/rfc/rfc5246.txt, accessed June 2015.
- [sha2] John Bryson, Patrick Gallagher, Approvers. "Secure Hash standards", FIPS 180-4. National Institute of Standards and Technology (NIST). March 2012. Available online: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf, accessed June 2015.
- [Sybard] Sybard® ICA Guard, Datasheet, QinetiQ, 2008. Available online: http://www.boldonjames.com/assets/downloadableFiles/sybard_ica_guard.pdf, accessed June 2015.
- [virtualbox] The VirtualBox User Manual, https://www.virtualbox.org/manual/UserManual.html, accessed June 2015 .
- [Zhiyong] C. Zhiyong et al., "Integrated Covert Channel Countermeasure Model in MLS Networks", IEEE International Conference on Information Engineering and Computer Science, pp. 1-4, 2009, Dec. 2009.

A. Traces of Authentication Verification Experiments

A.1. AuthNPassword Policy Traces

ARP request and response

```
13:22:17.631753 00:16:3e:44:f0:00 (oui Unknown) > Broadcast, ethertype
 Unknown (0x4305), length 64:
 0x0000:  0001 d1f0 060f 0001 0016 3e44 f000 7465  ..........>D..te
 0x0010:  7374 332e 4952 4154 492f 312f 2fff ffff  st3.IRATI/1//...
 0x0020:  ffff ff74 6573 7432 2e49 5241 5449 2f31  ...test2.IRATI/1
 0x0030:  2f2f                                     //
13:22:17.643269 00:16:3e:44:f0:93 (oui Unknown) > 00:16:3e:44:f0:00 (oui
 Unknown), ethertype Unknown (0x4305), length 64:
 0x0000:  0001 d1f0 060f 0002 0016 3e44 f093 7465  ..........>D..te
 0x0010:  7374 322e 4952 4154 492f 312f 2f00 163e  st2.IRATI/1//..>
 0x0020:  44f0 0074 6573 7433 2e49 5241 5449 2f31  D..test3.IRATI/1
 0x0030:  2f2f                                     //
```

M_CONNECT message

```
13:22:17.646113 00:16:3e:44:f0:00 (oui Unknown) > 00:16:3e:44:f0:93 (oui
 Unknown), ethertype Unknown (0xd1f0), length 164:
 0x0000:  99c6 ec95 3201 0000 1200 0100 0000 0000  ....2...........
 0x0010:  4000 9100 0000 0000 0873 1000 1801 2a00  @........s....*.
 0x0020:  3200 3800 4800 5000 9201 210a 1c50 534f  2.8.H.P...!..PSO
 0x0030:  435f 6175 7468 656e 7469 6361 7469 6f6e  C_authentication
 0x0040:  2d70 6173 7377 6f72 6412 0131 9a01 00a2  -password..1....
 0x0050:  010a 4d61 6e61 6765 6d65 6e74 aa01 0131  ..Management...1
 0x0060:  b201 0b74 6573 7432 2e49 5241 5449 ba01  ...test2.IRATI..
 0x0070:  00c2 010a 4d61 6e61 6765 6d65 6e74 ca01  ....Management..
 0x0080:  0131 d201 0b74 6573 7433 2e49 5241 5449  .1...test3.IRATI
 0x0090:  da01 00e0 0101
                      ......
```

Challenge request and response messages

```
13:22:17.765556 00:16:3e:44:f0:93 (oui Unknown) > 00:16:3e:44:f0:00 (oui
 Unknown), ethertype Unknown (0xd1f0), length 143:
 0x0000:  8dcf 4327 3201 0000 1100 0100 0000 0000  ..C'2...........
 0x0010:  4000 7c00 0000 0000 0800 100c 1800 2a11  @.|...........*.
 0x0020:  6368 616c 6c65 6e67 6520 7265 7175 6573  challenge.reques
 0x0030:  7432 1163 6861 6c6c 656e 6765 2072 6571  t2.challenge.req
 0x0040:  7565 7374 3800 4212 2a10 6661 3337 4a6e  uest8.B.*.fa37Jn
 0x0050:  6343 4872 7944 7362 7a61 4800 5000 9201  cCHryDsbzaH.P...
```

```
  0x0060:  020a 009a 0100 a201 00aa 0100 b201 00ba  ................
  0x0070:  0100 c201 00ca 0100 d201 00da 0100 e001  ................
  0x0080:  00                                       .
 13:22:17.766324 00:16:3e:44:f0:00 (oui Unknown) > 00:16:3e:44:f0:93 (oui
  Unknown), ethertype Unknown (0xd1f0), length 139:
  0x0000:  0261 afb2 3201 0000 1200 0100 0000 0000  .a..2...........
  0x0010:  4000 7800 0000 0000 0800 100c 1800 2a0f  @.x...........*.
  0x0020:  6368 616c 6c65 6e67 6520 7265 706c 7932  challenge.reply2
  0x0030:  0f63 6861 6c6c 656e 6765 2072 6570 6c79  .challenge.reply
  0x0040:  3800 4212 2a10 0d07 0302 2040 0272 7a41  8.B.*......@.rzA
  0x0050:  4d6a 1204 4a0a 4800 5000 9201 020a 009a  Mj..J.H.P.......
  0x0060:  0100 a201 00aa 0100 b201 00ba 0100 c201  ................
  0x0070:  00ca 0100 d201 00da 0100 e001 00         .............
```

M_CONNECT_R message

```
 13:22:17.770951 00:16:3e:44:f0:93 (oui Unknown) > 00:16:3e:44:f0:00 (oui
  Unknown), ethertype Unknown (0xd1f0), length 133:
  0x0000:  a792 b079 3201 0000 1100 0100 0000 0000  ...y2...........
  0x0010:  4000 7200 0000 0000 0873 1001 1801 2a00  @.r......s....*.
  0x0020:  3200 3800 4800 5000 9201 020a 009a 0100  2.8.H.P.........
  0x0030:  a201 0a4d 616e 6167 656d 656e 74aa 0101  ...Management...
  0x0040:  31b2 010b 7465 7374 332e 4952 4154 49ba  1...test3.IRATI.
  0x0050:  0100 c201 0a4d 616e 6167 656d 656e 74ca  .....Management.
  0x0060:  0101 31d2 010b 7465 7374 322e 4952 4154  ..1...test2.IRAT
  0x0070:  49da 0100 e001 01                        I......
 13:22:17.772301 00:16:3e:44:f0:00 (oui Unknown) > 00:16:3e:44:f0:93 (oui
  Unknown), ethertype Unknown (0xd1f0), length 154:
  0x0000:  ce3f 2c13 3201 0000 1200 0100 0000 0000  .?,.2...........
  0x0010:  4000 8700 0000 0000 0800 100e 1801 2a16  @.............*.
  0x0020:  656e 726f 6c6c 6d65 6e74 2069 6e66 6f72  enrollment.infor
  0x0030:  6d61 7469 6f6e 321e 2f64 6166 2f64 6166  mation2./daf/daf
  0x0040:  206d 616e 6167 656d 656e 742f 656e 726f  .management/enro
  0x0050:  6c6c 6d65 6e74 3800 420b 3209 0812 1203  llment8.B.2.....
  0x0060:  3130 3018 0048 0050 0092 0102 0a00 9a01  100..H.P........
  0x0070:  00a2 0100 aa01 00b2 0100 ba01 00c2 0100  ................
  0x0080:  ca01 00d2 0100 da01 00e0 0100
```

A.1.1. AuthNAssymetricKey (RSA) Policy Traces

ARP request and response

```
 19:17:39.606183 00:16:3e:44:f0:96 (oui Unknown) > Broadcast, ethertype
  Unknown (0x4305), length 64:
  0x0000:  0001 d1f0 060f 0001 0016 3e44 f096 7465  ..........>D..te
  0x0010:  7374 312e 4952 4154 492f 312f 2fff ffff  st1.IRATI/1//...
  0x0020:  ffff ff74 6573 7432 2e49 5241 5449 2f31  ...test2.IRATI/1
  0x0030:  2f2f                                     //
 19:17:39.617567 00:16:3e:44:f1:93 (oui Unknown) > 00:16:3e:44:f0:96 (oui
  Unknown), ethertype Unknown (0x4305), length 64:
  0x0000:  0001 d1f0 060f 0002 0016 3e44 f193 7465  ..........>D..te
  0x0010:  7374 322e 4952 4154 492f 312f 2f00 163e  st2.IRATI/1//..>
  0x0020:  44f0 9674 6573 7431 2e49 5241 5449 2f31  D..test1.IRATI/1
  0x0030:  2f2f
```

M_CONNECT message

```
 19:17:39.687501 00:16:3e:44:f0:96 (oui Unknown) > 00:16:3e:44:f1:93 (oui
  Unknown), ethertype Unknown (0xd1f0), length 451:
  0x0000:  0d52 3d19 3201 0000 1000 0100 0000 0000  .R=.2...........
  0x0010:  4000 b001 0000 0000 0873 1000 1801 2a00  @........s....*.
  0x0020:  3200 3800 4800 5000 9201 bf02 0a18 5053  2.8.H.P.......PS
  0x0030:  4f43 5f61 7574 6865 6e74 6963 6174 696f  OC_authenticatio
  0x0040:  6e2d 7373 6832 1201 311a 9f02 0a03 4544  n-ssh2..1.....ED
  0x0050:  4812 0641 4553 3132 381a 0453 4841 3122  H..AES128..SHA1"
  0x0060:  0764 6566 6175 6c74 2a80 02ae 4da1 2cda  .default*...M.,.
  0x0070:  2d89 e4ee bb77 9e7d 8ae3 0174 0268 83ae  -....w.}...t.h..
  0x0080:  480e e4d6 477b 24e9 14fb ad55 a507 c2b9  H...G{$....U....
  0x0090:  f04e 6231 8ac1 d023 563b 6e52 a993 2de7  .Nb1...#V;nR..-.
  0x00a0:  7e3b c6ba f3c9 e14d 48f2 62e3 72c1 6606  ~;.....MH.b.r.f.
  0x00b0:  94c9 f779 19fe 6732 a815 4191 971d c06c  ...y..g2..A....l
  0x00c0:  1455 0890 0f39 00fa 6fa0 ae2f 5103 a7c1  .U...9..o../Q...
  0x00d0:  db57 9b5f b6b9 92b5 2335 482a 5f14 49f6  .W._....#5H*_.I.
  0x00e0:  cf15 e135 c687 da2c d708 36a6 3f2d cb6f  ...5...,..6.?-.o
  0x00f0:  4c70 a837 632e 8c18 91cb 5ddb 8e2c 3267  Lp.7c.....]..,2g
  0x0100:  22f2 0a9f d293 2446 9429 2361 bd6c 9141  ".....$F.)#a.l.A
  0x0110:  e42c 52a3 6f91 d723 675f 99e0 e77b dd00  .,R.o..#g_...{..
  0x0120:  985a 0f42 d00b 5622 8e25 8c58 f19e 150e  .Z.B..V".%.X....
  0x0130:  9baa f26a 2dc1 7cc7 e898 2381 922b 11f3  ...j-.|...#..+..
  0x0140:  038d 5409 c828 cd14 7c73 1f46 4e4c 1fbb  ..T..(..|s.FNL..
  0x0150:  28e9 40d8 9954 7584 71bf 0c8d 5887 1271  (.@..Tu.q...X..q
  0x0160:  4142 d5ca d5e4 4b77 29bb ea9a 0100 a201  AB....Kw).......
  0x0170:  0a4d 616e 6167 656d 656e 74aa 0101 31b2  .Management...1.
  0x0180:  010b 7465 7374 322e 4952 4154 49ba 0100  ..test2.IRATI...
  0x0190:  c201 0a4d 616e 6167 656d 656e 74ca 0101  ...Management...
  0x01a0:  31d2 010b 7465 7374 312e 4952 4154 49da  1...test1.IRATI.
  0x01b0:  0100 e001 01
```

EDH exchange and encrypted client challenge message

```
 19:17:39.797199 00:16:3e:44:f1:93 (oui Unknown) > 00:16:3e:44:f0:96 (oui
  Unknown), ethertype Unknown (0xd1f0), length 441:
```

```
  0x0000:  4666 18a0 3201 0000 1100 0100 0000 0000  Ff..2...........
  0x0010:  4000 a601 0000 0000 0800 100c 1800 2a21  @.............*!
  0x0020:  4570 6865 6d65 7261 6c20 4469 6666 6965  Ephemeral.Diffie
  0x0030:  2d48 656c 6c6d 616e 2065 7863 6861 6e67  -Hellman.exchang
  0x0040:  6532 2145 7068 656d 6572 616c 2044 6966  e2!Ephemeral.Dif
  0x0050:  6669 652d 4865 6c6c 6d61 6e20 6578 6368  fie-Hellman.exch
  0x0060:  616e 6765 3800 429b 0232 9802 0a03 4544  ange8.B..2....ED
  0x0070:  4812 0641 4553 3132 381a 0453 4841 3122  H..AES128..SHA1"
  0x0080:  002a 8002 b6da 1287 fcbc 9614 0c0f 422d  .*............B-
  0x0090:  e740 10ab 8d07 1832 f2ac baab 5540 7b90  .@.....2....U@{.
  0x00a0:  2835 eaf8 f167 294b fd0c db8c 073a b637  (5...g)K.....:.7
  0x00b0:  6d4b 263c 38a5 1243 88e5 08f0 2691 b845  mK&<8..C....&..E
  0x00c0:  fc7c f2eb 5721 b007 7e7d c60c f05d e17d  .|..W!..~}...].}
  0x00d0:  9c49 ee56 e358 2317 3284 7651 4358 88a9  .I.V.X#.2.vQCX..
  0x00e0:  9cff 0bd9 c9be 783c 7ceb 4721 27db d2ec  ......x<|.G!'...
  0x00f0:  71de 20f6 c660 a906 e4c7 4988 aaa3 1096  q....`....I.....
  0x0100:  0af3 433d 6d81 bed6 bafa 93aa 425f 140a  ..C=m.......B_..
  0x0110:  41af d44e 6814 76b6 0681 5877 af63 68bc  A..Nh.v...Xw.ch.
  0x0120:  5131 9f19 2aae bae5 ab7a d447 c3cd 1815  Q1..*....z.G....
  0x0130:  f86a 7498 5155 1cc8 9e29 22d3 7b10 fd53  .jt.QU...)".{..S
  0x0140:  00b4 592f 4bb2 0a50 cacf 49bc bfd9 2d18  ..Y/K..P..I...-.
  0x0150:  3997 6950 1736 cc4a ccd1 7291 5608 89d0  9.iP.6.J..r.V...
  0x0160:  c670 e04e da72 7d3f 0685 5701 4d7d 3839  .p.N.r}?..W.M}89
  0x0170:  3ef8 9d78 6022 dc1c 1737 3268 e014 e914  >..x`"...72h....
  0x0180:  6259 4a5e 4800 5000 9201 020a 009a 0100  bYJ^H.P.........
  0x0190:  a201 00aa 0100 b201 00ba 0100 c201 00ca  ................
  0x01a0:  0100 d201 00da 0100 e001 00
 19:17:39.833505 00:16:3e:44:f0:96 (oui Unknown) > 00:16:3e:44:f1:93 (oui
  Unknown), ethertype Unknown (0xd1f0), length 386:
  0x0000:  30b2 24db 161b 631f ec4d 67ad 44a0 8675  0.$...c..Mg.D..u
  0x0010:  4297 76c6 e94e 40f6 6617 4d2c bf8e 7b5e  B.v..N@.f.M,..{^
  0x0020:  b812 0309 7d3b 9d36 e8db 857d fd6f bb40  ....};.6...}.o.@
  0x0030:  7b65 c478 20ee 26ac 83d8 5137 7671 d0eb  {e.x..&...Q7vq..
  0x0040:  8f94 0e0e 5714 bd0e 54e9 e9e6 e6ca ebe7  ....W...T.......
  0x0050:  c766 4ae2 fce6 898e a26b 9237 9454 3e75  .fJ......k.7.T>u
  0x0060:  94c1 cda8 29dc c0da 42e4 6139 2c74 a4cb  ....)...B.a9,t..
  0x0070:  406c 03cc d861 953f 1077 b33a 197e ecee  @l...a.?.w.:.~..
  0x0080:  f008 231d 0849 b72c 0f40 2ad6 00ff 8f42  ..#..I.,.@*....B
  0x0090:  b921 eec6 9b39 9612 b0ba ff73 624f b948  .!...9.....sbO.H
  0x00a0:  7356 2d11 fd9d 2f9b 2d35 43d3 28fb 32df  sV-.../.-5C.(.2.
  0x00b0:  3d07 3dfd f36f 878c 7139 bf81 8792 afe2  =.=..o..q9......
  0x00c0:  4b3a 2852 f114 1fc6 c1a7 b41b e821 7cd3  K:(R.........!|.
  0x00d0:  a8ce cfbc 9482 862a a92e 3bda b0c6 06b2  .......*..;.....
  0x00e0:  fac4 d8b2 05e7 b30e 7dfb f17b 10ee 44cb  ........}..{..D.
  0x00f0:  ade6 162d 98bf c843 de6e c70f 0d07 d731  ...-...C.n.....1
  0x0100:  2194 253e 8858 ca53 29af c0f4 a7b2 3607  !.%>.X.S).....6.
  0x0110:  b589 f711 ecbc ec87 50f2 d072 f91f 6d8a  ........P..r..m.
  0x0120:  6d3d b99e a5ea f43b 29ce 7653 6f9e a079  m=.....;).vSo..y
  0x0130:  e28e b885 cae4 36eb 03d8 0458 fb17 afdc  ......6....X....
  0x0140:  7997 dac9 4b87 801f e77a a373 6c00 46cc  y...K....z.sl.F.
  0x0150:  5f9c c00a 54ef 0e8f e3b1 54dd a8fc 07f6  _...T.....T.....
  0x0160:  d165 5233 9126 dc9b 0b38 8385 2770 2dd4  .eR3.&...8..'p-.
  0x0170:  b349 0783
```

IPCP *test1.IRATI* log

```
3242(1433265459)#librina.cdap-manager (DBG): Waiting timeout 180000 to
 receive a connection response
3242(1433265459)#ipcp[2].routing-ps-link-state (DBG): flow allocation
 waiting for enrollment
3242(1433265459)#ipcp[2].rib-daemon (DBG): Received CDAP message through
 portId 1:
12_M_WRITE
Object class: Ephemeral Diffie-Hellman exchange
Object name: Ephemeral Diffie-Hellman exchange
Object value: 0xf4a034d0
Scope: 0

3242(1433265459)#librina.security-manager (DBG): Generated encryption key
 of length 16 bytes: 3ef06968c6f8698d6ed037ff4f197d62
3242(1433265459)#ipcp (DBG): Requesting the kernel to enable encryption on
 port-id: 1
3242(1433265459)#librina.nl-manager (DBG): NL msg RX. Fam: 25; Opcode:
 42_ENABLE_ENCRYPT_RESP; Sport: 0; Dport: 3242; Seqnum: 1433265397;
 Response; SIPCP: 2; DIPCP: 0
3242(1433265459)#librina.nl-manager (DBG): NL msg TX. Fam: 25; Opcode:
 41_ENABLE_ENCRYPT_REQ; Sport: 3242; Dport: 0; Seqnum: 1433265397;
 Request; SIPCP: 2; DIPCP: 2
3242(1433265459)#librina.core (DBG): Added event of type
 41_ENABLE_ENCRYPTION_RESPONSE and sequence number 1433265397 to events
 queue
3242(1433265459)#librina.core (DBG): Waiting for message 3242
3242(1433265459)#rinad.event-loop (DBG): Got event of type
 41_ENABLE_ENCRYPTION_RESPONSE and sequence number 1433265397
3242(1433265459)#librina.security-manager (DBG): Encryption and decryption
 enabled for port-id: 1
3242(1433265459)#librina.syscalls (DBG): Invoking SYS_writeManagementSDU
 (361)
3242(1433265459)#ipcp[2].rib-daemon (DBG): Sent CDAP message of size 345
 through port-id 1:
12_M_WRITE
Object class: Client challenge
Object name: Client challenge
Object value: 0x93427b0
Scope: 0

3242(1433265459)#librina.syscalls (DBG): Invoking SYS_readManagementSDU
 (360)
3242(1433265459)#ipcp[2].rib-daemon (DBG): Received CDAP message through
 portId 1:
12_M_WRITE
Object class: Client challenge reply and server challenge
Object name: Client challenge reply and server challenge
Object value: 0xf4a034d0
Scope: 0

3242(1433265459)#librina.security-manager (INFO): Remote peer successfully
 authenticated
3242(1433265459)#librina.syscalls (DBG): Invoking SYS_writeManagementSDU
 (361)
3242(1433265459)#ipcp[2].rib-daemon (DBG): Sent CDAP message of size 115
 through port-id 1:
12_M_WRITE
Object class: Server challenge reply
Object name: Server challenge reply
Object value: 0xf4a02e08
Scope: 0

3242(1433265459)#librina.syscalls (DBG): Invoking SYS_readManagementSDU
 (360)
3242(1433265459)#librina.cdap-manager (DBG): Connection response received
3242(1433265459)#ipcp[2].rib-daemon (DBG): Received CDAP message through
 portId 1:
1_M_CONNECT_R
Abstract syntax: 115
Authentication policy: Policy name: PSOC_authentication-ssh2
Supported versions: 1
Source AP name: test2.IRATI
Source AP instance: 1
Source AE name: Management
Destination AP name: test1.IRATI
Destination AP instance: 1
Destination AE name: Management
Invoke id: 1
Result: 0
Version: 1
```

B. Updated LFA Policy

B.1. Narrative description of the Loop Free Alternates policy

B.1.1. The Flow State Database

The Flow State Database is the subset of the RIB that contains all the Flow State Objects (FSOs) known by the IPC Process. It is used as an input to calculate the Routing Table. The FSDB consists of the operations on FSOs received through CDAP messages.

B.1.1.1. RIB Objects:

Flow State Object (FSO)

The object exchanged between IPC Processes to disseminate the state of one N-1 flow supporting the IPC Processes in the DIF. This is the RIB target object when the PDU Forwarding Table Generator wants to send information about a single N-1 flow.

```
../fsdb/<address>/<neighbour_address>/<QoS> : flowstateobject
    address          /* The address of the IPC Process */
    neighbour_address /* The address of the neighbour IPC Process */
    QoS-cube         /* The QoS of this N-1 flow */
```

B.1.2. Routing Table

Based on the FSDB, a graph of the connectivity in the DIF is constructed. From this graph, a routing table can be calculated for every QoS cube in the DIF. However, in this specification, only the shortest route is calculated using Dijkstra, using hop count as the metric for distance. Apart from this, for every node, the Loop Free Alternates are also calculated. Node Protecting Loop Free Alternates are preferred over Link Protecting Loop Free Alternates. An example connectivity graph is shown in Figure B.1, “An example connectivity graph”, and its corresponding routing table as calculated by A is shown in Table B.1, “Routing table of IPC process with address A”. Note that from A to B there are 2 N-1 flows with different QoS.

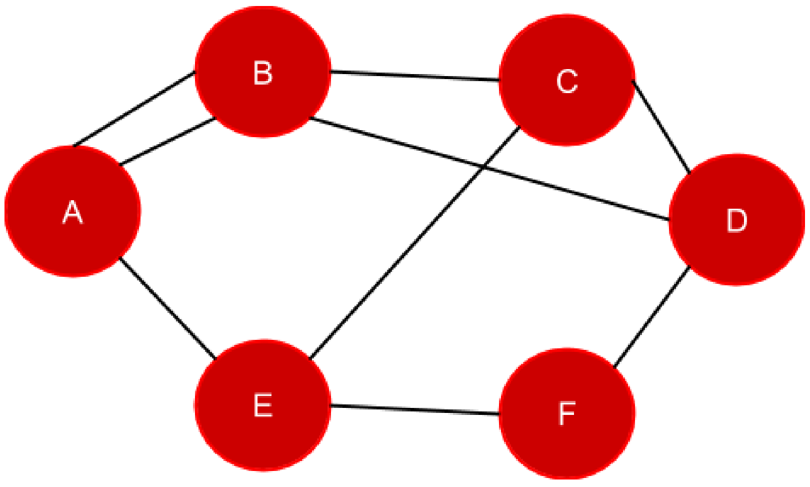

Figure B.1. An example connectivity graph

Table B.1. Routing table of IPC process with address A

Destination Address	Next Hop	LFA
B	B	B
C	B	E
D	B	E
E	E	B
F	E	B

B.1.3. PDU Forwarding Table

Based on the routing table, the PDU forwarding table is calculated in each node. In essence, this is the mapping of the next hop on a port-id. In the example, suppose there are 2 flows to B from A, with port-id 1 and 2, and there is one flow from A to E with port-id 3. Then a generated forwarding table could look as follows:

Table B.2. Forwarding table of IPC process with address A

Destination Address	Port-id	LFA
B	2	1
C	2	3
D	1	3
E	3	1
F	3	2

This table is then consulted by the Relaying and Multiplexing Task (RMT) to decide on what port-id the PDU should be written.

B.1.4. Subscription and reaction to events

Upon initialization of the PFT, the PFT subscribes to certain events of the RIB daemon. This makes the PDU Forwarding Table Generator completely event based. The cooperation between these tasks in the IPC process is depicted in Figure B.2, “Cooperation of tasks in the IPC process”. These events are:

- N-1 flow up
- N-1 flow down
- Flow State Database has changed

Apart from subscribing to these events, the PFT marks all objects in the FSDB to be replicated upon changes.

B.1.4.1. N-1 flow up

When invoked

This is an event that indicates an N-1 flow is up again.

Action upon receipt

If there is a Delete_FSO timer corresponding with this flow, it is stopped. Else, a Flow State Object is created, containing the address of the IPC process and the address of the neighbour IPC process where the flow is allocated to. The QoS is set to the QoS of the flow. The FSO is added to the FSDB unless there is already an FSO present with the same addresses and the same QoS.

B.1.4.2. N-1 flow down

When invoked

This is an event that indicates an N-1 flow to a neighbour is down.

Action upon receipt

The Delete_FSO timer is started on this flow. Note that this time should be chosen reasonably small.

B.1.4.3. Delete_FSO expires

When invoked

This is invoked when the Delete_FSO timer fires.

Action upon receipt

The Flow State Object corresponding with this flow is deleted, unless there is another neighbour flow with the same addresses and QoS present in the IPC process. If the port-id of the flow is present in the forwarding table, the LFA is used until a new forwarding table is generated.

B.1.4.4. Flow State DB has changed

When invoked

This is an event that indicates there was a change to the Flow State Database.

Action upon receipt

Upon this event, the routing table is re-calculated. If there is already a calculation on-going it is stopped and restarted. After the routing table has been calculated, the forwarding table is generated from it.

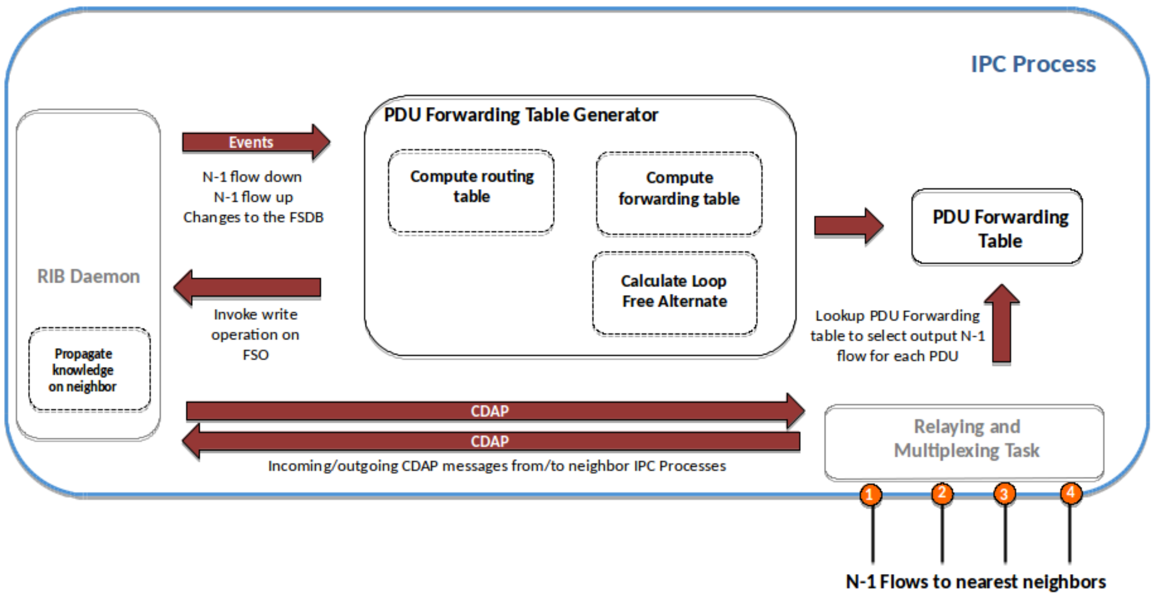
IPC Process
RIB Daemon
Propagate knowledge on neighbor
Events
N-1 flow down
N-1 flow up
Changes to the FSDB
Invoke write operation on FSO
PDU Forwarding Table Generator
Compute routing table
Compute forwarding table
Calculate Loop Free Alternate
PDU Forwarding Table
Lookup PDU Forwarding table to select output N-1 flow for each PDU
CDAP
CDAP
Incoming/outgoing CDAP messages from/to neighbor IPC Processes
Relaying and Multiplexing Task
1
2
3
4
N-1 Flows to nearest neighbors

Figure B.2. Cooperation of tasks in the IPC process

C. Updated FLD Policy

Flow Liveness Detection (FLD) detects if a flow between IPC processes is alive or not by sending periodic messages. When FLD is present, the Flow Manager keeps two additional states for the flow - i.e. UP and DOWN. FLD maintains a timer that is reset upon reception of such a periodic message. The flow is declared DOWN if the timer expires, otherwise it is declared UP.

C.1. Common elements

The procedures described in the remaining sections, rely on the following common elements:

FLD elements:

```
Keepalive:
    Timeout : Timer

FLD data:
    port-id : Port-id
    keepalive : Keepalive
    interval : Int (milliseconds)

RIB objects:

../fld/<neighbour-address>-<address>/<connection-id>
Timeout : Double

../fld/<address>-<neighbour-address>/<connection-id>
Timeout : Double
```

A RIB object containing a timeout value - i.e. ../fld/<neighbour-address>-<address>/<connection-id> - is periodically updated with a new timeout value on each corresponding CDAP M_WRITE. FLD subscribes to changes to this object and is thus notified when it has been changed. The Keepalive timer is then restarted with the new timeout value. If the Keepalive timer expires the FLD notifies the FMGR that the flow is DOWN.

C.2. Initialization

The Timeout value for the Keepalive timer has to be chosen depending on the DIF. Most likely it will be a function of the Round Trip Time (RTT). For initialization of the FLD, the following steps are followed:

- Firstly, FLD will subscribe to changes to the RIB object ../fld/<address>-<neighbour-address>/<connection-id> through the RIB Daemon, where <connection-id> is the connection-id that identifies the flow with the peering IPC process.
- Secondly, FLD will ask the RIB Daemon to periodically, every Interval milliseconds, replicate ../fld/<neighbour-address>-<address>/<connection-id> to the peer's RIB.
- Finally, the Keepalive timer is started.

C.3. FLD Behaviour

C.3.1. Keepalive_Timer.expire

When invoked

Whenever the Keepalive timer expires.

Action upon invocation

The FMGR is notified that the flow should be declared DOWN.

C.3.2. Timeout_Changed.receive

When invoked

Upon changes to ../fld/<address>-<neighbour-address>/<connection-id>

Action upon receipt

The Keepalive timer is re-armed with the communicated timeout value. Communicating a 0 timeout is allowed and implies declaring the flow as DOWN immediately. This could be used for interrupting incoming traffic without deallocating the flow.