Deliverable-4.3

Final specification and consolidated implementation of security and reliability enablers

Deliverable Editor: Dimitri Staessens, iMinds VZW

Publication date:	30-June-2016
Deliverable Nature:	Report
Dissemination level (Confidentiality):	PU (Public)
Project acronym:	PRISTINE
Project full title:	PRogrammability In RINA for European Supremacy of virTuaIIsed NEtworks
Website:	www.ict-pristine.eu
Keywords:	Security, DIF, DAF, IPC Process, access control, authentication, SDU protection, resiliency
Synopsis:	D4.3 describes the final 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: Dimitri Staessens, iMinds VZW
iMINDS: Dimitri Staessens, Sander Vrijders
i2CAT: Eduard Grasa, Berta Delgado
TSSG: Miguel Ponce de Leon, Micheal Crotty, Ehsan Elahi
FIT-BUT: Ondrej Rysavy, Vladimir Vesely, Ondrej Lichtner
IMT: Ichrak Amdouni, Anis Laouiti, Hakima Chaouchi
PNSol: Peter Thompson, Neil Davies
TRT: Sarah Haines, Hamid Asgari
NXW: Vincenzo Maffione
BISDN: Victor Alvarez

Executive Summary

The main security requirements for consideration in information systems are specified as: to prevent unauthorized information disclosure (confidentiality) and improper malicious modifications of information (integrity), while ensuring access for authorized entities (availability). In RINA's architecture design, programmability, the ability to dynamically configure, control and combine the security functions to address the above requirements from cyber security, cyber and network resiliency aspects, are regarded as the key features to create a secure, reliable, and resilient network system. This document, D4.3, builds upon the security and resiliency functions, mechanisms, and techniques that are described in D4.1 and their enhancements reported in D4.2. This report provides further and mainly final developments of security and resiliency components within PRISTINE's WP4 in terms of specification, implementation, verification, and validation. These functions, mechanisms and techniques include the Authentication of IPC processes, Cryptographic function, Access Controls (Capability-Based Access Control and Multi-Level Security), and Key Management system. The resiliency and high availability aspects including the work on resilient routing, whatever-cast and load balancing are reported. This document also reports on the work carried out on formal verification of security threats identified and reported in D4.1. This deliverable overall provides the relevant specifications and analysis, the design aspects, Proof of Concept implementations (PoC), and related PoC tests using IRATI stack. Specifically, we report on the following aspects in relation to the security and resiliency functions specified above:

- The architecture and specification of relevant functions and their designs whenever any enhancement made and is deemed necessary to report
- The use-case scenarios and options for application of specified security/ resiliency functions/enablers
- The relevant policies and options to show the programmability where each designed security component can function based on the providers/ users needs
- The implementation and realisation of components, their interfaces and interactions

- The verification and validation tests carried out to further prove the correct functionality of components.

Future enhancement will be carried out due to the extension of project's duration especially for the work on Key Management System and will be reported at the end of the project. The implemented security functions and enablers are being ported and the work will be continued in WP6 for system-level integration, verification, validation tests and experimentation in use-case scenarios. Given the above aspects, in WP4 we enhanced, realised and demonstrate some functionalities established in the architectural structure of RINA in providing an inherently secure and reliable networking environment.

Table of Contents

Acronyms 11
1. Introduction 13
1.1. Specification and System Design 13
1.2. Implementation Tasks 13
2. Authentication of IPC Processes 15
2.1. New authentication policy: AuthTLSHandshake 16
2.1.1. Specification 17
2.1.2. Implementation 22
2.1.3. Validation 23
2.2. Update to the SSH2 Authentication policy 30
3. Cryptographic Functions and Enablers 32
3.1. SDU Protection Final Implementation 32
3.1.1. SDU Protection TTL Policy Set 34
3.1.2. SDU Protection Cryptographic Policy Set 34
3.1.3. SDU Protection Error Check Policy Set 35
3.2. SDU Protection Policy Configuration Specification 36
3.3. Evaluation and Experiments 37
4. Capability-based Access Control 40
4.1. Introduction 40
4.2. Background 40
4.3. Proposed CBAC architecture 42
4.3.1. Overview 42
4.3.2. Architecture 43
4.3.3. Token Structure 44
4.3.4. Token Generation and Access Control 45
4.4. Access Control Scenarios in RINA 46
4.4.1. Enrolment Scenario 46
4.4.2. Operation of Layer Management Functions Scenario 47
4.5. Integration and Specification of the Proposed CBAC architecture in RINA 47
4.5.1. Updates on the RINA Management System for CBAC Integration 48
4.5.2. Access Control Scenarios in a RINA DIF with CBAC 49
4.5.3. Specification of the Enrollment Scenario 50
4.6. Implementation of the CBAC Policy in the IRATI Stack 52
4.6.1. CBAC Interaction with Other IRATI Components 52

4.6.2. CBAC with Authentication Policy 53
4.6.3. Technologies Used in CBAC Implementation 53
4.6.4. Proof of Concept Scenario 54
4.7. Annex: AC threats and countermeasures 64
4.7.1. Enrollment scenario 65
4.7.2. Scenario 2: RIB Access 66
5. Multi-Level Security 68
5.1. MLS in RINA 68
5.2. BPC Phases 69
5.2.1. Initialisation Phase 70
5.2.2. Operational Phase 70
5.2.3. Monitoring Phase 71
5.3. BPC Specification and Design 71
5.3.1. BPC Architecture Option 1: BPC at an AP 72
5.3.2. BPC Architecture Option 2: BPC split between an AP and an IPCP 74
5.3.3. BPC Architecture Option 3: BPC at an IPCP 76
5.4. Implementation of BPC 78
5.4.1. SDU Format 78
5.4.2. Boundary Protection Policy 79
5.5. Verification of BPC Function 81
5.5.1. Experimentation Environment 81
5.5.2. Verification Tests and the Results 85
6. Key Management System: specification 91
6.1. Considerations for Key Management 91
6.1.1. Time-of-day related issues 92
6.1.2. Legal intercept, validation and conformance testing 93
6.1.3. Bootstrapping and rebooting 93
6.1.4. Interworking between domains 94
6.1.5. Bulk encryption 94
6.2. Key management architecture 94
6.2.1. Communication between the CKM and the KMAs 95
6.2.2. Relationship to RIB 96
6.2.3. Storage of key material 96
6.2.4. Stateless operation 97
6.2.5. Entropy 97
6.3. Use cases 97
6.3.1. Generation and distribution of keys 98

6.3.2. Periodic rotation of keys 99
6.3.3. Password authentication 102
6.3.4. AuthNAsymmeticKey Policy 103
7. Formal verification of security controls 107
7.1. Definition of Threat Model 107
7.2. RINA Network Model 118
7.3. Security Analysis 123
8. Resiliency and High Availability 128
8.1. Resilient Routing Policies 128
8.1.1. Specification of the Loop Free Alternates (LFA) routing policy 128
8.1.2. Implementation in IRATI 132
8.1.3. Validation 135
8.1.4. Conclusions 142
8.2. Management of names in Networking 143
8.2.1. Introduction 143
8.2.2. Glossary 143
8.2.3. Applications in RINA 145
8.2.4. Application Name-space Management 146
8.2.5. Processing System Name-space Management 147
8.2.6. Distributed Application Facility Name-space Management 148
8.2.7. Root Name-space Management 150
8.3. Providing Resiliency through Whatever-cast 151
8.3.1. Proposed whatever-cast scheme 152
8.3.2. Detailed working in every layer 156
8.3.3. Implementation in RINASim 157
8.4. Load balancing 161
8.4.1. Introduction 161
8.4.2. Load Balancing in RINA 162
8.4.3. Option 1: Pseudo Random Selection 164
8.4.4. Option 2: Minimum Hop Selection (Geographical Load Distribution) 165
8.4.5. Option 3: Delegation by a proxy DAP 166
8.4.6. Option 4: Redirection using Client AP 168
8.4.7. Redirection using a DAF policy 170
8.4.8. Conclusion and Future Work 171
8.5. Providing Parallel flows over the shim DIFs 172

8.5.1. Specification for the Shim DIF over Ethernet with Link Layer Control (LLC) 173
8.5.2. Shim DIF over UDP with DNS 183
8.5.3. Specification for the Shim DIF over IPv4/UDP with Domain Name System (DNS) Support 183
8.5.4. Implementation 193
8.5.5. Validation 195
8.5.6. Conclusion 197
9. Summary and Conclusions 198
References 201
A. Traces of Authentication Verification Experiments 205
B. Log files from MLS Verification Tests 213
C. Code for formal verification of security controls 219

List of Figures

1. Workflow of the Auth TLS policy 17
2. TLS authentication policy verification scenario 24
3. Avg RTT 29
4. Throughput 30
5. Goodput of the default and crypto SDU Protection Policies 38
6. Relative comparison of goodput, packet rate and RTT of default and crypto SDU Protection Policies 39
7. Proposed CBAC architecture. 44
8. Updates on the RINA management system for CBAC integration 48
9. Access Control Scenarios in a RINA DIF with CBAC 49
10. Specification of AC in enrollment scenario in RINA 51
11. Specification of AC in operation of layer management functions scenario 52
12. IPCP components that interact with the CBAC implementation 52
13. Experimental scenario 54
14. BPC at the Application Level 73
15. BPC split between AP and IPCP 75
16. BPC at the DIF-level 77
17. Experimentation Environment 82
18. Experimentation Environment for Verification Tests 1 and 2 82
19. Experimentation Environment for Verification Test 3 83
20. Experimentation Environment for Verification Test 4 84
21. key management function architecture 95
22. Creation and storage of a new public/private key pair 99
23. Rotation of a key 101
24. Password Authentication using Key Management Agent 102
25. Password Authentication using Key Management Agent Cache 103
26. Password Authentication using Key Management Proxy 103
27. Password Authentication using Asymmetric Key Part 1 104
28. Password Authentication using Asymmetric Key Part 2 105
29. A Model of Two Hosts over a Direct Channel 109
30. Formal Specification of Simplified Scenario 110
31. A Network Topology of Analysed Scenario 119
32. A RINASim Simulation Model 127
33. RINASim PDUViewer showing the content of captured PDU 127
34. An example connectivity graph 129
35. Cooperation of tasks in the IPC process 132

36. Topology for which original LFA implementation was incorrect ... 133
37. Experiment scenario for LFA fast-reroute 136
38. An application within a processing system. 145
39. Namespace management within an application process. 147
40. Namespace management within a processing system. 148
41. Namespace management within a Distributed Application Facility. 150
42. Global name-space management. 151
43. Reference scenario for whatever-cast resiliency 153
44. Resiliency Factor parameter 158
45. Directory illustration 159
46. Neighbour table illustration 160
47. Whatever-cast RINASim topology 160
48. Load Distribution in RINA 163
49. Server DAP Ai acting as proxy node between client and the available server DAP Aj 167
50. Server Ai rejects with a new suggestion which is told to the client. .. 169
51. Server Ai rejects with a new suggestion. 170
52. Relation between protocol machines 176
53. State diagram of the shim DIF 183
54. State diagram of the shim DIF 193

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
CKL	Compromised Key List
CRC	Cyclic Redundancy Check
CRL	Certificate Revocation List
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
HSM	Hardware Security Module
IPC	Inter Process Communication
IPCM	Inter Process Communication Manager
IPCP	Inter Process Communication Process
IRATI	"Investigating RINA as an Alternative to TCP/IP" project
KA	Key Agent
KFA	Kernel Flow Allocator
KM	Key Manager

KMF	Key Management Function
KMIP	Key Management Interoperability Protocol
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
XKMS	XML Key Management Specification
XML	eXtensible Markup Language

1. Introduction

This deliverable provides the final specifications, design, and implementations of innovative security functions and reliability enablers developed within the PRISTINE project. It covers the functions and enablers described in D4.1 and D4.2 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 final implementations of these mechanisms and functions; addressing the security and reliability requirements of the scenarios analysed in D2.1.

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 and resiliency. WP2 deliverables described the overall PRISTINE reference architecture. Deliverables D4.1 and D4.2 provided the overall PRISTINE functional security architecture and specifies each of the main security functions and the interactions among them, as well as the reliability mechanisms. This deliverable presents the final specifications and their implementation.

The software developed as part of this deliverable is available on the IRATI GitHub pages.

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 and resiliency objectives. Deliverables 4.1 and 4.2 provided the RINA security solution, the functions and the relevant enablers to achieve the above objectives. The functions and enablers for security include: Authentication, Access Control, Secure Channel and SDU Protection, Key Management functions, monitoring and countermeasures for reducing the security risks and combating the threats. The functions and enablers for resiliency include Failure

Detection, Resilient Routing and leveraging the Whatever-cast mechanism for reducing the impact of link (n -1 flow) and node (IPCP) failures.

2. Authentication of IPC Processes

With the aim of continuing the work started in D4.2 [D4.2], the authentication task of WP4 has kept exploring the space of potential authentication policies that can be relevant to a DIF environment. A number of such potential approaches were initially identified in D4.1 [D4.1], and a few of them thoroughly specified and implemented as reported in D4.2.

During the last phase of the project the authentication work has been focused on certificate-based authentication. Certificate-based authentication relies on digital certificates, an electronic document that binds an identity with a public key. Regardless of its limitations, the use of certificates helps prevent the use of fake public keys for impersonation. Digital certificates are issued by trusted third parties called Certificate Authorities (CAs), which sign all the certificates they issue with their private key. A Certificate Authority may be a Root CA - the root of trust for a particular chain - or may rely on an upstream CA who has previously delegated the rights to issue certificates to it. If so, validating a digital certificate requires validating the whole chain of certificates until the certificate of the root CA is reached (the root CA signs its certificate with its own private key, instead of having an upstream CA signing it). Expiration dates built-in the digital certificates and lists of certificates that have been revoked (certificate revocation lists) maintained by CAs provide tools to minimize the lifetime of bogus or misused certificates.

The most popular application of digital certificate authentication is the Transport Layer Security (TLS) framework, heavily used by HTTPS to enable secure transactions over the web such as online banking. TLS is in fact two protocols: the **TLS Handshake protocol** [RFC5246], which performs authentication and negotiates the algorithms and keys to be used by the **TLS Record protocol** [RFC5246] to perform encryption, compression and message integrity validation functions. WP4 has adapted the cryptographic operations and information exchange performed by the TLS Handshake protocol as an authentication policy for CACEP, thus allowing the use of a well-known certificate-based authentication procedure in the DAF/DIF environments. Cryptographic mechanisms in the TLS Record protocol have also been mined and adapted as SDU

protection policies - this work is reported in the next section of this deliverable.

2.1. New authentication policy: AuthTLSHandshake

This authentication policy is based on the cryptographic procedures of the TLS Handshake protocol. The correct operation of this policy depends on the mandatory use of the TLSRecord SDU Protection policy, which is based on the TLS Record policy. It is an example of an authentication policy that uses certificates, in which each IPC Process has its own X.509 digital certificate.

The handshake protocol consists of a sequence of messages sent between IPC processes. This protocol is used to negotiate the set of algorithms for authentication, confidentiality, compression and integrity; generate shared secrets, crypto keys and negotiate other parameters for the session. It allows peers to authenticate each other. In the end, these security parameters must be provided to the record layer, which is configured with the algorithms and keys negotiated by the handshake protocol. The differences of the current version of the CACEP authentication policy with respect to the normal TLS Handshake protocol are the following:

- No support of session resuming (useful feature foreseen for future extensions of the policy).
- No support for HelloRequest by "server".
- No support for extensions.
- Client is always obliged to provide its certificate for authentication.
- Limited availability of cipher suites. In the first version there will be only support for one cipher suite to demonstrate the correct operation of the policy.
  - *RSA* as key exchange algorithm (implies no need for ServerKeyExchange message)
  - *RSA* as a client certificate signing algorithm (no need for CertificateRequest message)
  - *AES128* or *AES256* as encryption algorithms (always used in *hard cbc* mode)
  - *MD5* or *SHA256* as message integrity verification algorithms (always used in *HMAC* mode)

- *deflate* as the compression algorithm

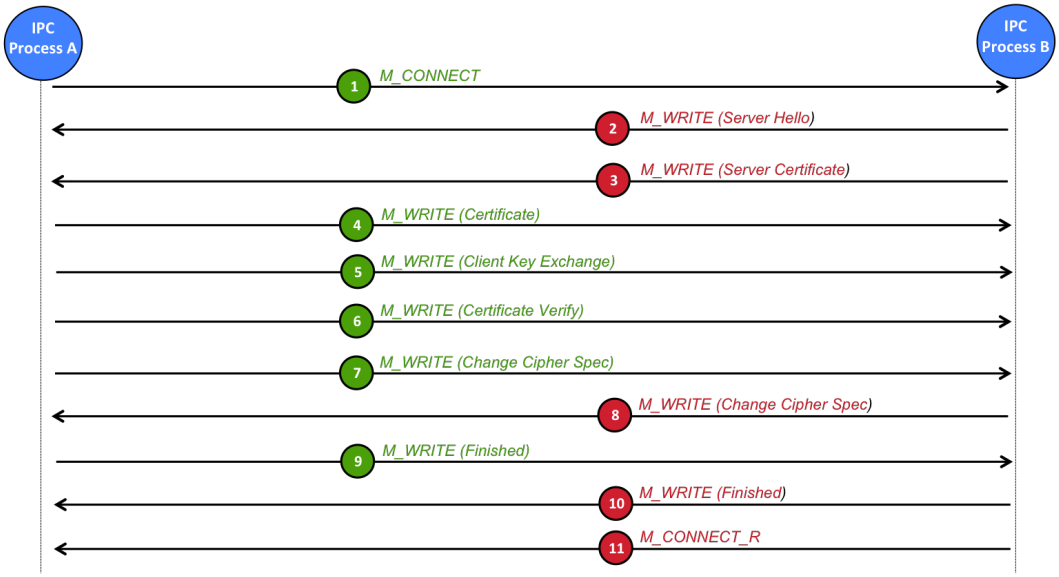

Figure 1. Workflow of the Auth TLS policy

2.1.1. Specification

When IPCP A first connects to IPCP B, it is required to send M_CONNECT (Client Hello) as its first message. With this message starts the application connection set-up procedure, it contains the list of configuration options that IPCP A can support. It looks as seen below:

IPCP A to IPCP B, M_CONNECT (equivalent to Client Hello)

- **Name**: PSOC_authentication-tlshandshake
- **Versions**: 1 (only supported version as of now).
- **Options**:
  - *ap_con_id*: If not zero, ID of the application connection to be resumed.
  - *Ciphersuites*: The list of cipher suites supported by the requesting IPCP, sorted by preference.
  - *Compression*: The list of compression methods supported by the requesting IPCP, sorted by preference.
  - *UTCUnixTime*: The current time and date in standard UNIX 32-bit format.
  - *RandomBytes*: 28 bytes generated by a secure random number generator.

IPCP B receives the M_CONNECT message and decides if the authentication policy is correct. If it is, it selects a single cipher suite and compression method from the list provided by IPCP A. If none are supported or the application connection is rejected, IPCP B sends an M_CONNECT_R indicating error. Otherwise it sends the following Server Hello message back to IPCP A:

IPCP B to IPCP A, M_WRITE(Server Hello)

- **Opcode**: M_WRITE
- **Object class**: Server Hello.
- **Object value**:
  - *Version* : The version of the policy chosen for the application connection
  - *ap_con_id*: The id of the application connection (for resuming the application connection)
  - *Ciphersuite*: The cipher suite chosen for the application connection
  - *Compression*: The compression method chosen for the application connection
  - *UTCUnixTime*: The current time and date in standard UNIX 32-bit format
  - *RandomBytes*: 28 bytes generated by a secure random number generator (independent from the ones generated by the client)

Once the negotiating process has ended, IPCP B needs to show its credentials to IPCP A, so that IPCP A can authenticate him. To do so, it sends its certificate chain. Certificate must be of type X.509v3, IPCP B's certificate public key must be compatible with the key exchange algorithm. Normally there are multiple options available in TLS, we choose to stick to the use of *RSA* as the key exchange algorithm (the certificate must allow the key to be used for encryption). Its Certificate chain is sent back to the "client" using the following message:

IPCP B to IPCP A, M_WRITE(Server certificate)

- **Opcode**: M_WRITE
- **Object class**: Server Certificate.

- **Object value**:
  - *Certificate chain* : The certificate chain of IPCP B. First is the certificate of IPCP B, then all the certificates that certify each other until one that provides a root of trust within the DIF.

Since the only supported key agreement algorithm is RSA and the client always needs to provide a certificate, there is no need for the "ServerHelloDone" message. When IPCP A receives the "ServerCertificate" message, it can already send a "Certificate" message to IPCP B, since no more messages from IPCP B are expected. The certificate must be of type X.509v3. The public key of IPCP A's certificate must be allowed to be used for signing with the signature scheme and hash algorithm that will be employed in the certificate verify message. The following diagram shows the next three messages that IPCP A needs to send to IPCP B.

IPCP A to IPCP B, M_WRITE(Client Certificate)

- **Opcode**: M_WRITE
- **Object class**: Certificate.
- **Object value**:
  - *Certificate chain* : The certificate chain of IPCP A. First is the certificate of IPCP A, then all the certificates that certify each other until one that provides a root of trust within the DIF.

After sending this message, IPCP A sends a ClientKeyExchange message. To do so, IPCP A generates 48 random bytes and encrypts them using the RSA public key extracted from Server certificate message. Once IPCP B has received them, it must decrypt the message with its RSA private key. The message contains the RSA-encrypted pre-master secret and looks like following:

IPCP A to IPCP B, M_WRITE(ClientKeyExchange)

- **Opcode**: M_WRITE
- **Object class**: Client Key Exchange.
- **Object value**:
  - *Encrypted pre-master secret* : The pre-master secreted generated by IPCP A, encrypted with the public key of IPCP B's certificate.

After this message, both parties have agreed upon the same pre-master secret. Then, both need to generate the master secret. The master secret is computed by using a PRF(pseudo random function) defined in [RFC5246], Chapter *5*. It is based on HMAC and always uses the SHA-256 hash algorithm.

```
master_secret = PRF(pre_master_secret, "master secret", ClientHello.random
 + ServerHello.random);
```

Both parties will compute the same master secret if the pre-master secret has been well decrypted. At this point IPCP B and IPCP A are able to compute the encryption keys. They need to calculate two keys for encryption (Rx and Tx) and two for the HMAC hash (Rx and Tx) operations. To do so, we use the same PRF function mentioned before with the following parameters, until enough bytes of key material are generated.

```
key_material = PRF(master_secret, "key expansion", ClientHello.random +
 ServerHello.random);
encrypt_key_tx = key_material[0, encrypt_key_length - 1];
encrypt_key_rx = key_material[encrypt_key_lenght, 2*encrypt_key_length
 -1];
hmac_key_tx = key_material[2*encrypt_key_length, 2*encrypt_key_length +
 hmac_key_length -1];
hmac_key_rx = key_material[2*encrypt_key_length + hmac_key_length,
 2*encrypt_key_length + 2*hmac_key_length -1];
```

The next message is used to verify the certificate of IPCP A. All the *ObjectValue* field of all the messages exchanged between IPCP A and IPCP B (up to and without including this message) need to be hashed with the SHA-256 algorithm. Then, they will be signed with IPCP A's private key and send it to IPCP B with the next message:

IPCP A to IPCP B, M_WRITE(CertificateVerify)

- **Opcode**: M_WRITE
- **Object class**: Certificate Verify
- **Object value**:
  - *Signed structure* : Hashes of the *ObjectValue* field of all messages exchanged between both parties, signed with the private key of IPCP A.

IPCP B also needs to compute the hash of the *ObjectValue* field of all messages exchanged between IPCP A and IPCP B (up to and without including this message) with the SHA-256 algorithm. Once its computed and IPCP B has received the *CertificateVerify* message, it must decrypt it using IPCP A's public key. If the authentication is done correctly, the computed and received hash should be the same. If not, an error message is generated (M_CONNECT_R with error code).

Once the security parameters are in place, IPCP A can send the *ChangeCipherspec* message to signal IPCP B that this is the last message before starting to use the new cipher suite. This message consist on a single byte of value 1. It is used to notify the other party (IPCP B) to start using the policies agreed during the handshake. Reception of this message causes IPCP B to enable the new SDU protection policies on the reception path over the N-1 flow. Once this is done IPCP B sends the *ChangeCipherSpec* message to IPCP A. IPCP A receives this message and enables the new SDU protection policies on both the reception and transmission paths over the N-1 flow. All subsequent messages are protected with the newly negotiated security parameters. IPCP B will enable the new policies on the transmission path when it receives the *Finished* message from IPCP A.

A *Finished* message is always sent after a *ChangeCiphserspec* message to verify that all SDU Protection policies are correctly in place. It contains the new configuration, and it is the first one protected with the just negotiated algorithms, keys, and secrets. To generate the finish message, IPCP A uses the same PRF function as explained before with the following parameters:

IPCP A to IPCP B, M_WRITE(Finished)

- **Opcode**: M_WRITE
- **Object class**: Client Finish
- **Object value**:
  - *Verify data* : PRF(master_secret, "finish label", hash(object_value field of all handshake_messages)).

IPCP B also computes the PRF function with its parameters. Once IPCP B receives the verify data compares it with its generated verify data, they should be equal if the content is correct. If not, an error message is generated. If verify data is the same in both parties, IPCP B can send its *Finished* message to IPCP A following the same procedure.

At the end of the authentication process a M_CONNECT_R message must be send. This message indicates that the authentication process has ended successfully.

2.1.2. Implementation

The TLS authentication policy specified has been implemented in librina, so that the policy 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 4.2 deliverable, 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. Configuration of the SDU Protection module requires asynchronous messaging (via Netlink sockets). This authentication policy uses OpenSSL's libcrypto library as it needs to perform reliable cryptographic operations. The libcrypto facilities used in this policy are: HMAC, RAND_BYTES and SHA256 functions, load RSA keys from PEM files, RSA public key encryption and private key decryption and RSA private key encryption and public key decryption.

TLSHandshake authentication policy inherit from the IAuthPolicySet abstract class, and therefore overwrites its pure virtual methods. **get_auth_policy** is the first operation implemented, is invoked by the Enrolment Task when it has to initiate the application connection. In this operation we obtain the values for the AuthPolicy field of the CDAP M_CONNECT message. It returns an AuthPolicy object. The **initiate_authentication** policy checks for the correct policy names and version, selects the algorithms to be used for encryption, and sends the SERVER_HELLO and SERVER_CERTIFICATE messages with their corresponding fields.

Once this is done, we call the **process_incoming_message_function** whenever an authentication message is received. This function processes the different messages involved in this policy:

- **Server Hello message**. IPCP A stores cipher suites, random bytes, compression algorithms and version in its security context.

- **Server certificate message**. IPCP A stores IPCP B's certificates for later use. Once it has receive this message, IPCP A can proceed to send its credentials.
- **Client certificate message**. IPCP B stores IPCP A's certificate.
- **Client key exchange**. After IPCP A generates the master secret and encrypts it using IPCP B's RSA public key, IPCP B receives it and decrypts the pre-master secret with its private RSA key. Both generate the master secret using the PRF function explained in the policy design section, in this operation they also compute the encryption keys.
- **Client certificate verify message**. During all previous message exchanges both IPCPs have hashed the *ObjectValue* field of sent/received messages using the SHA256 algorithm. IPCP A has sent its verification hash signed with its RSA private key. In this operation IPCP B decrypts it using IPCP A's public key, and compares the verification hash received with its own computation. If the authentication has been correct the verification hash must be equal.
- **Client Change Cipher Spec message**. When this message is received IPCP B requests to enable the negotiated SDU protection policies on the receive path for the related N-1 port in the kernel.
- **Server Change Cipher Spec message**. When this message is received, IPCP A requests to enable the negotiated SDU protection policies on the receive and transmit paths for the related N-1 port in the kernel.
- **Client Finished message**. IPCP B calculates the verification data and compares it with the received one. If they are equal, the procedure has been successful. Then, IPCP B request to enable the negotiated SDU protection policies on the transmit path for the related N-1 port in the kernel.
- **Server Finished message**. IPCP A compares the received verification data with its own computation. If the result is equal, the operation returns SUCCESS.

2.1.3. Validation

Authentication policy verification scenario

The experimental scenario used to verify the correct operation of the AuthTLSHandshake policy is shown in the verification scenario figure.

A normal DIF consisting of two IPCPs operates over one shim DIF over Ethernet. IPCP test1.IRATI and IPCP test2.IRATI are configured to use the AuthTLSHandshake policy by default, with encryption, Error Check (CRC) and TTL policies.

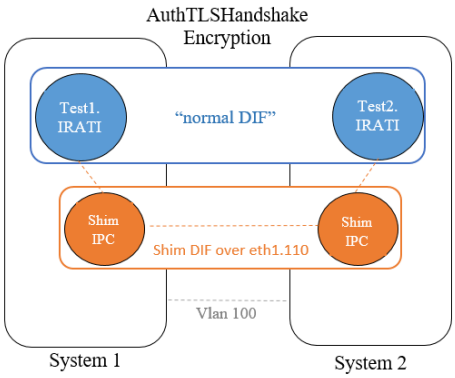

Figure 2. TLS authentication policy verification scenario

Configuration

The following code snippet shows an example of the AuthTLSHandshake policy configuration, as well as of the associated encryption policy that must be activated for N-1 port. The authentication policy needs to be populated with the location of the certificates files and the RSA private key.

```
"securityManagerConfiguration" : {
         "policySet" : {
           "name" : "default",
           "version" : "1"
           },
           "authSDUProtProfiles" : {
               "default" : {
                  "authPolicy" : {
                    "name" : "PSOC_authentication-tlshandshake",
                "version" : "1",
                "parameters" : [ {
                     "name" : "keystore",
                     "value" : "/usr/local/irati/etc/creds-tls"
                    }, {
                     "name" : "keystorePass",
                     "value" : "none"
                    } ]
                },
                "encryptPolicy" : {
                   "name" : "default",
```

```
                "version" : "1",
                "parameters" : [ {
                  "name" : "encryptAlg",
                  "value" : "AES128"
                 }, {
                  "name" : "macAlg",
                  "value" : "SHA256"
                 } , {
                  "name" : "compressAlg",
                  "value" : "deflate"
              } ]
            },
            "TTLPolicy" : {
             "name" : "default",
             "version" : "1",
             "parameters" : [ {
                "name" : "initialValue",
                "value" : "50"
              } ]
            },
            "ErrorCheckPolicy" : {
               "name" : "CRC32",
               "version" : "1"
            }
          }
        }
    },
```

The files contained by the folder provided as the value as the "keystore" parameter have to be arranged in the following way:

- **key**: contains the RSA private key of the IPCP, in PEM format.
- **cert.pem**: contains the certificate of the IPCP, in PEM format.
- <**cert_name**>**.pem**: one PEM file for every certificate required to reach the (or one of the) root of trust of the DIF (root CA).

Certificate generation

One of the first steps to run TLS authentication is to create the certificates needed for the policy. Although there can be used other trustier certificates we explain one way to generate them for testing using self-signed certificates (this should never be done in a production scenario). This method consists on creating a ROOT Certificate Authority (CA) from an

OpenSSL's default script and two self-signed certificates from OpenSSL commands.

We will work from the directory: /etc/ssl We move the CA.pl script (use locate CA.pl if you can't find it) to the current directory and execute *./CA.pl -newca* in order to start creating the root certificate. After this command, we have generated a public-private key pair. We can find the public key at */etc/ssl/ca/cacert.pem* and the private key at */etc/ssl/ca/private/cakey.pem*. To obtain this CA certificate is necessary to set a password, as it is needed for signing the end-user certificate.

```
root@debian:/etc/ssl# ./CA.pl -newca
CA certificate filename (or enter to create)

Making CA certificate ...

... More output ...

Write out database with 1 new entries
Data Base Updated
```

CA will use a CSR (certificate signing request) to create our SSL certificate. It contains information that will be included in the end-user certificate such as organization name, location and country. It also contains the public key that will be included in the end-user certificate. At the same time, we create the private key of the end-user certificate, we need to keep the private key "secret". To do so we have used the following command, the key generated is a RSA key of 2048 bits.:

```
root@debian:/etc/ssl# openssl req -nodes -new -newkey rsa:2048 -keyout
 demoCA/private/cert1key.pem -out demoCA/certs/cert1csr.pem
Generating a 2048 bit RSA private key
.....................................+++

... More output ...

An optional company name []:
```

This creates a new CSR (in etc/ssl/demoCA/certs/cert1csr.pem) which must then be signed using the CA's private key, and a private key (in /etc/ssl/ca/private/cert1key.pem). Last step is to create and sign the end-user

certificate. We will take the input CSR (demoCA/certs/cert1csr.pem) and create the end user certificate (in demoCA/certs/cert1.pem). The command line used is:

```
root@debian:/etc/ssl# openssl ca -policy policy_anything -in demoCA/certs/
cert1csr.pem -out demoCA/certs/cert1.pem
Using configuration from /usr/lib/ssl/openssl.cnf
Enter pass phrase for ./demoCA/private/cakey.pem:
Check that the request matches the signature

... More output ...

1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

This resulting certificate will be the one used for the policy. Then we will need a second certificate for IPCP B. We will follow the same procedure, creating a new CSR and a new end-user certificate to obtain the second certificate.

Traces of authentication Verification experiments

The following traces are the output of capturing the Ethernet packets at the eth1.110 interfaces in system 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 test1.IRATI request a flow allocation to the IPC Process test2.IRATI.

M_CONNECT message reflects test1.IRATI sending an M_CONNECT messages to test2.IRATI, requesting a new connection to be opened using the "PSOC_authentication-tlshandshake" authentication policy.

IPCP test2.IRATI replies with the Server Hello and Server Certificate message Server hello and server certificate messages. Now, test1.IRATI sends its client certificate message, client key exchange message and client certificate verify message Client certificate, client key exchange and client certificate verify messages. The last non-encrypted message is client change cipher spec message, and its corresponding response, server change cipher spec Client and server Change Cipher Spec messages.

From now on, all messages are encrypted, as shown by the trace of the Finished messages Encrypted client and server Finished messages. Since the communication is encrypted, showing the log of tcpdump of the following packets is not very illustrative. The last trace 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 sent and received the messages, until it receives an M_CONNECT_R message indicating that the application connection has been successfully established.

Performance analysis

In this section we will analyse the performance of this new policy and compare it with the use of no authentication. The results obtained in the measurement of the enrolment time are very variable, we have computed the average with 10 samples. It takes about **16,4 ms** for TLS handshake authentication policy to enrol, a not very significant increase over the **8 ms** of the use of no authentication - which involves no cryptographic operations and 8 messages less to exchange. The reason is that both machines are virtual machines co-located in the same physical machine, therefore since the RTT is very low the cost of exchanging a message is also low. With longer RTTs the cost of message exchanges will dominate over the cost of cryptographic operations, increasing the enrolment time in relation to that of a policy with no authentication.

The second set of measurements we have taken consist on calculating the round trip time (RTT) of an application over an encrypted N-1 flow. To do so, we have used a script called rina-echo-time that pings the other IPCP and computes the minimum, maximum and average time, as well as the standard deviation, we show the results of the average RTT in the following figure:

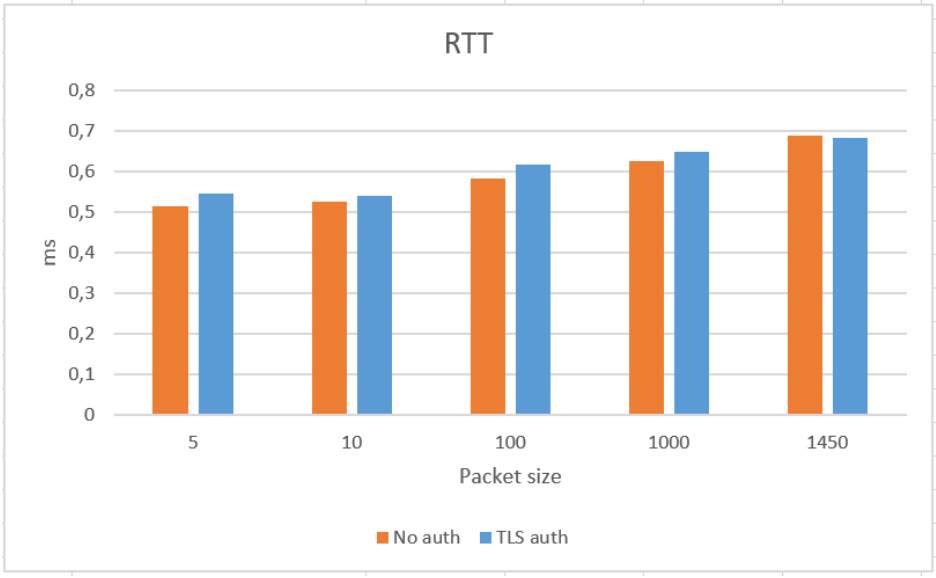

Figure 3. Avg RTT

Figure above is computed sending each time 10000 packets of variable size. The maximum size established has been 1450 bytes for packet since it is the maximum size to avoid fragmentation; a feature not supported yet. We can observe that the RTT is just moderately impacted by the use of the TLS handshake and associated SDU protection policies. It has a higher RTT when the packets are small but, as we increase the size of the packets we obtain approximately the same results as in the use of no authentication.

To conclude this section we will show the throughput study results in the Figure below. We have used the traffic generator application to obtain them. This application generates traffic during a specific period of time with a predefined packets size. Our period of time will be 20 second and the same sizes as in the RTT analysis. As we increase the size of the packets we observe that the overhead of performing crypto operations per PDI compromises significantly the throughput.

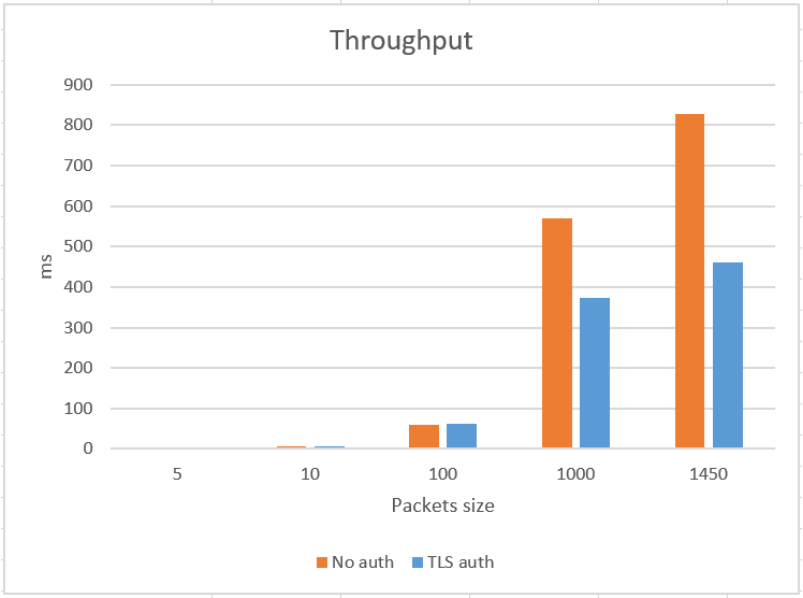

Figure 4. Throughput

2.2. Update to the SSH2 Authentication policy

During the experimentation phase after D4.2 publication an error/ vulnerability in the first implementation of the SSH2 Authentication policy was identified: the policy was using a single RSA key pair for both authenticating parties, which doesn't justify the use of public key cryptography and is not how SSH operates. During the months previous to the publication of D4.3 the implementation has been updated so that each IPCP has its own RSA key pair. As a consequence, each IPCP needs to know the public keys of the IPCPs it wants to establish an application connection to.

The code is slightly changed, as well as the configuration of this policy. Now the "keystore_path" configuration parameter expects a path to a folder (terminating without "/"). The contents of the folder should be the following files:

- **key**: The RSA private key of the IPCP in PEM format (generated for example with openssl genrsa -out key 2048)
- **public_key**: The RSA public key of the IPCP in PEM format (extracted from the private key file, for example with openssl rsa -in key -pubout > mykey.pub)
- For each IPCP whose public key is known, a file containing his public key in PEM format. The file must be named with the IPC Process application

process name. For example, imagine the IPCP has 3 neighbours, then there would be 3 files: *B.IRATI*, *C.IRATI* and *D.IRATI*.

3. Cryptographic Functions and Enablers

The SDU Protection module is a part of the IPC Process (IPCP) data path. The SDU protection is applied on a per-port basis, that means, each unique flow can have different SDU protection policy or configuration. Outgoing SDUs are protected by applying protective functions. Incoming SDUs are checked by applying corresponding verification functions. By that way, the 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. The simplest possible mechanism can be null policy representing no operation applied on SDU in none of the directions.

SDU protection involves functionality that applies various 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.

According to the overall RINA specifications, SDU protection can perform a 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.

3.1. SDU Protection Final Implementation

The implementation of SDU Protection described in D4.2 modified the SerDes kernel module to add a limited functionality of protecting the transferred data against external threats. This limited implementation was provided as a Proof of Concept for the placement and feasibility of the full SDU Protection implementation. This section describes the current state of implementation of the SDU Protection kernel component.

The final implementation of the SDU Protection module stands as an independent kernel module connected to the RMT module. This module is independent of the SerDes module unlike the PoC implementation. The various functions applied by SDU Protection were separated into three

distinct policy sets that can be configured/replaced independently of each other. They are:

1. **SDU Protection Lifetime Limiting Policy Set**, implementing lifetime limiting policies
2. **SDU Protection Cryptographic Policy Set**, implementing compression, replay detection, data integrity, and data encryption policies
3. **SDU Protection Error Check Policy Set**, implementing data transmission error checking policies

The SDU Protection module implements the mechanism of selecting a specific policy set implementation for each (N-1)-port used by the current IPC Process. The details of the interfaces of the individual policy sets needed for development of custom implementations will be described later in this document.

In addition to managing policy sets for open (N-1)-ports, the SDU Protection module connects to the RMT with the following functions:

```
int sdup_protect_pdu(struct sdup_port * instance, struct pdu_ser * pdu);
int sdup_unprotect_pdu(struct sdup_port * instance, struct pdu_ser * pdu);
int sdup_set_lifetime_limit(struct sdup_port * instance, struct pdu_ser *
 pdu, struct pci * pci);
int sdup_get_lifetime_limit(struct sdup_port * instance, struct pdu_ser *
 pdu, size_t * ttl);
int sdup_dec_check_lifetime_limit(struct sdup_port * instance, struct pdu
 * pdu);
```

Where the `sdup_protect_pdu` and `sdup_unprotect_pdu` functions handle calling the respective protect and unprotect functions of the Cryptographic and Error Check policy sets. The remaining lifetime limit related functions directly export the interface of the Lifetime Limiting policy set. The separation from the module wide protect/ unprotect functions is a result of limitations of the current IPC Process implementation, where the SDU Protection module cannot access the deserialized form of the PDU to store an intermediate value needed by the policy set.

3.1.1. SDU Protection TTL Policy Set

In contrast with other SDU Protection policies, the TimeToLive value is a parameter associated with the N-DIF (unlike other policies that depend on the (N-1)-DIF) and is carried over from one protected PDU to the next one, during the PDUs data path in a relaying IPCP. Because of this this policy set implements three policies:

```
int sdup_set_lifetime_limit_policy(struct sdup_ttl_ps *ps, struct pdu_ser
 *pdu, size_t pci_ttl);
int sdup_get_lifetime_limit_policy(struct sdup_ttl_ps *ps, struct pdu_ser
 *pdu, size_t *ttl);
int sdup_dec_check_lifetime_limit_policy(struct sdup_ttl_ps *ps, struct
 pdu *pdu);
```

Where the `sdup_{set, get}_lifetime_limit_policy` functions add/retrieve the lifetime limit value to/from the protected PDU. And the `sdup_dec_check_lifetime_limit_policy` performs the liveness check and decrementation of the intermediate lifetime limit value during the relaying process.

The current implementation of this policy set implements the well known TTL mechanism. During the transmission of the PDU, over the (N-1)-flow, a TTL value is placed at the beginning of the serialized PDU, effectively creating a `TTLProtectedPDU`, this is represented by a `struct pdu_ser` structure that was extended to include the TTL value at the beginning. Every time the PDU is relayed by a member of the N-DIF, this value decreased by 1, and the PDU is discarded when the value reaches 0.

3.1.2. SDU Protection Cryptographic Policy Set

This policy set implements the following three policies:

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

Where the `sdup_{apply, remove}_crypto` functions perform cryptographic protection/unprotection of the transmitted PDU, and the `sdup_update_crypto_state` function can be used to interact with the

Security Manager component to manage the cryptographic state of the policy set, for e.g. to change the encryption keys.

The current implementation of the protection and unprotection mechanisms extends the functionality of the PoC implementation by adding missing operations. It is inspired by the **DTLS Record Protocol** to provide an equivalent level of security services, namely data integrity, data confidentiality and limited replay detection. During protection the policy set performs the following operations:

1. compression
2. addition of sequence number
3. addition of an HMAC signature
4. PKCS7 padding
5. encryption

Similarly, the unprotection mechanism performs the inverse operations in the reverse order:

1. decryption
2. check and removal of PKCS7 padding
3. check and removal of HMAC signature
4. removal of sequence number and check for replay
5. decompression

The compression, HMAC signature, encryption and their inverse operations use the Linux Crypto API algorithm implementations. This means that even though the current implementation is limited to a specific set of algorithms it can be easily extended to support any other algorithm that is already provided by the Crypto API. The current implementation supports:

- AES128, AES256 in CBC mode for encryption
- MD5, SHA256 hash algorithms for HMAC
- Deflate algorithm for compression

3.1.3. SDU Protection Error Check Policy Set

This policy set implements the following two policies:

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

Where the `sdup_add_error_check_policy` function generates an error check value and adds it to the protected PDU, and the `sdup_check_error_check_policy` function retrieves this value and performs the related mechanism that detects and optionally repairs any errors caused by PDU transmission.

The current implementation of this policy uses the CRC32 mechanism (same as in the PoC implementation). In addition to being separated into an independent policy set module, the current implementation changed the format of the resulting protected PDU. The calculated CRC value is now placed at the end of the input PDU to create the `ErrProtectedPDU` (represented by a `struct pdu_ser` instance). This is to more closely resemble the industry standard of placing error protection codes at the end of data streams, which simplifies implementation in hardware.

3.2. SDU Protection Policy Configuration Specification

Configuration of parameters related to SDU Protection is part of the Security Manager section of the DIF configuration. The following is the relevant part of the configuration:

```
"securityManagerConfiguration" : {
    ...
    "authSDUProtProfiles" : {
        "default" : {
            ...
            "encryptPolicy" : {
                "name" : "default",
                "version" : "1",
                "parameters" : [
                    {
                        "name" : "encryptAlg",
                        "value" : "AES256"
                    },{
                        "name" : "macAlg",
                        "value" : "SHA256"
                    }, {
```

```
                        "name" : "seq_win_size",
                        "value" : "30"
                    }, {
                        "name" : "compressAlg",
                        "value" : "deflate"
                    }]
            },
            "TTLPolicy" : {
                "name" : "default",
                "version" : "1",
                "parameters" : [
                    {
                        "name" : "initialValue",
                        "value" : "50"
                    }
                ]
            },
            "ErrorCheckPolicy" : {
                "name" : "CRC32",
                "version" : "1"
            }
            ...
        }
    }
    ...
}
```

The parameters parsed from the policy configurations are used by the Security Manager during Enrollment parameter negotiation. After successful authentication and parameter negotiation, the Security Manager sends the result to the SDU Protection module in the kernel part of the IPC Process. In case of the Cryptographic protection policy set the data is carried by an **RINA_C_IPCP_UPDATE_CRYPTO_STATE_REQUEST** Netlink message, that gets translated into a call of the `sdup_update_crypto_state` function call of the related policy set. This message carries information about the negotiated cryptographic algorithms and the keys that are to be used for encryption and message authentication.

3.3. Evaluation and Experiments

The implementation of SDU protection policy was evaluated in several trials that were provided in deliverable D6.2. Here, we present results of

performance evaluation of Crypto SDU Protection Policy implementation. This measurement identifies processing overhead of cryptographic operation of SDU protection policy. The relative performance was compared only in the case when the cryptographic operations were not accelerated by the hardware support. The test environment consisted of the two RINA systems virtualized using XEN VMs hosted in the same physical machine. The Ethernet drivers for the virtual NICs were utilized without bandwidth restriction. The graph in Figure 1 shows the total performance in the experimental environment. As it can be seen, the crypto SDU Protection Policy achieves significantly worse results.

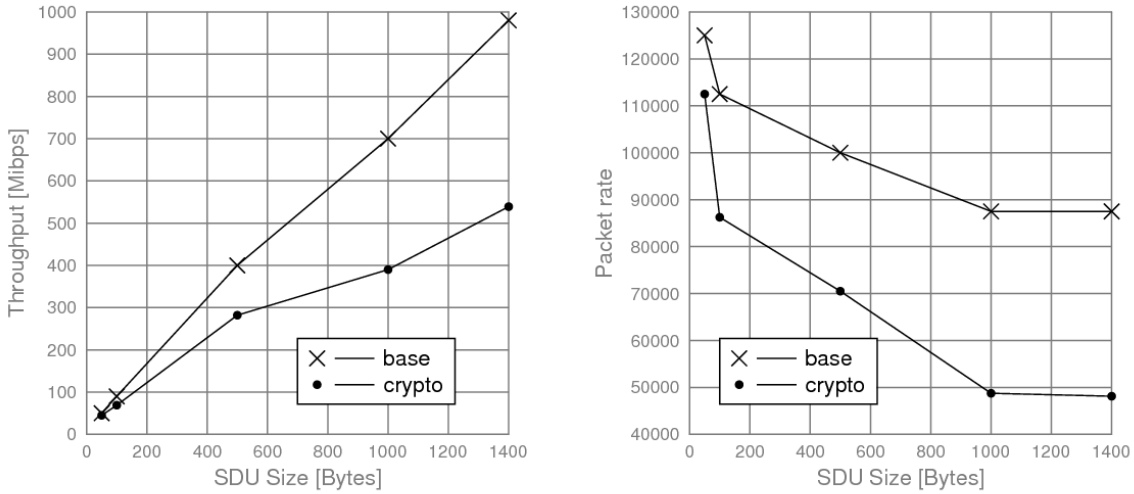

Figure 5. Goodput of the default and crypto SDU Protection Policies

Figure 2 shows how the RTT and goodput degrade when using the cryptographic SDU protection policies. The baseline is represented by the basic SDU protection policy. This protection does only CRC computation and TLL. Presented results show how performance degrades when crypto policy is applied. In other words, the overhead of crypto SDU protection operations such as encryption, integrity hash computation is measured.

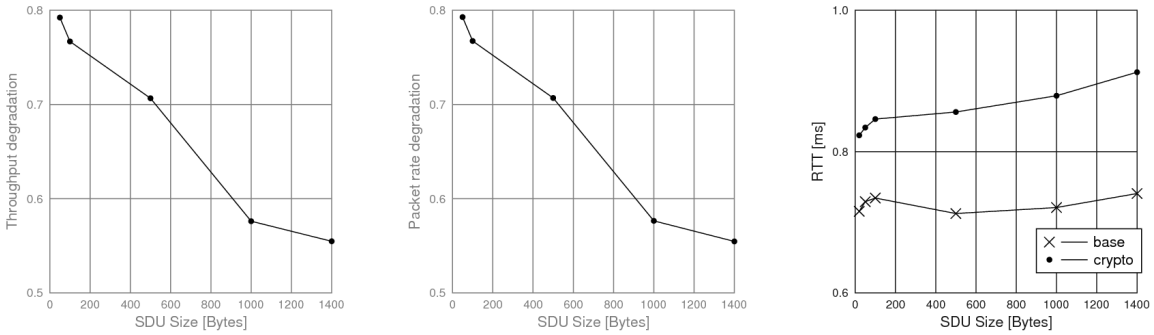

Figure 6. Relative comparison of goodput, packet rate and RTT of default and crypto SDU Protection Policies

Without encryption, RTT is mostly independent of the SDU size, but the graph shows how encryption introduces a non-negligible processing cost per SDU that causes the RTT to grow with the SDU size. Regarding goodput, performance drops up to 60% for the case of directly connected IPCPs when encryption is used. The overhead introduced by the encryption can be reduced if the cryptographic operations are supported by the hardware, for instance, for the hardware platform equipped with Advanced Encryption Standard Instruction Set.

4. Capability-based Access Control

4.1. Introduction

Authorisation and Access Control (**AC**) refer to the concepts of allowing a set of **subjects** (e.g. users, processes, roles, entities) accessing **objects** (e.g. data, files, contents). Authorisation defines whether the subject should be allowed to access objects. Access control manages this access at a very granular level. Compared to the authentication, the AC provides a deeper level of the system security with respect to the resources of the system. Indeed, the authentication is the process of verifying whether the claimed identity is the real one; its output is usually yes or no. The AC checks the rights of the requester regarding the requested resources. Authentication is usually a necessary step for AC but it is not the unique step. The main objective of AC is to prevent unauthorised information disclosure (confidentiality) and improper malicious modifications (integrity), while ensuring access for authorised entities (availability). An access control policy specifies the conditions under which subjects are authorised to have access to objects.

Access Control in PRISTINE is a crucial step that must be performed in various cases where different requesters (subjects) would like to access to different resources (objects). In a DIF, IPCPs are considered as subjects that are required to be authorised to proceed with some actions on the objects. Objects are the data and contents within the DIF of which the IPCPs are members. Basically, the CBAC policy will provide the corresponding capabilities to allow IPCPs to get access to the required resources in the DIF. We consider two main scenarios: enrolment in a DIF and operation of layer management functions.

To address the AC problem is PRISTINE, we propose **CBAC**; a capability based access control architecture. This architecture is integrated in RINA as a policy for the security management component. We will show how this architecture profits from RINA design to define fine grained and flexible access control rules. CBAC is also implemented in the IRATI stack.

4.2. Background

In computer networks, access control is a key task to secure the network *control plane*. Indeed, computer networks are composed of multiple layers

of protocol machines cooperating to perform a common task: allowing instances of distributed applications to communicate. The cooperating network protocol machines themselves need to exchange information to coordinate while performing distributed functions such as routing or resource allocation. In this context, authentication and access control are two key security components to prevent malicious attackers. As such, it is desirable to restrict the ability to perform coordination functions to the protocol machines that need to execute them; applying the principle of less privilege. These coordination functions are usually known as the control plane, and are typically implemented via multiple protocols. For example routing can be performed by the OSPF, IS-IS or BGP protocols; each one defining its own operations and data objects. Enforcing network access control rules to the IP control plane is usually done by the use of simple Access Control Lists (ACLs) [rfc6192], which will block certain ports where control plane protocols are running from certain IP addresses. Stateful L3/L4 firewalls are also used. They inspect the contents of the control plane protocol headers [jun-patent]. The port checking mechanism is simple but very rigid. It is also not very secure, since only access to a port is checked but not the contents of the protocol header. The stateful L3/L4 firewall mechanism is more flexible; a node could be allowed to perform read-style operations but not write-style ones. However, this mechanism is very expensive: new code in the firewall should be added. Last but not least, the firewall itself is a middle-box that has to be deployed and managed, and also introduces a performance penalty in terms of delay. More flexible and fine-grained access control mechanisms are required in order to efficiently manage the access control issue in computer networks. In the literature there are various AC models [benantar]. For instance, Role Based Access Control (RBAC) [rbac] categorise users based on similar needs and group them into roles. Permissions are assigned to roles rather than to individual users. Its objective is to reduce the number of assignments. Like the ACLs, RBAC suffers from scalability issues especially in a distributed context and extended cross domain environments. Furthermore, RBAC does not permit to represent dynamic data such as location, time and date. Attribute based Access control (ABAC) [abac] method was introduced as a new solution to these mentioned issues. ABAC defines the authorisation based on the evaluation of certain attributes which include the attributes of the entities (subjects and objects) that require the access to a certain resource in the system and the related environment. While RBAC provides

coarse-grained, predefined and static access control configurations, ABAC offers fine-grained rules which are evaluated dynamically in real time. This makes ABAC more flexible and permits to consider the dynamic context in the access decision. Despite its numerous advantages, the ABAC model still have scalability issues and needs accordingly an effective definition of the attributes.

Capability-based AC approach (CBAC) [cbac], [cbacIoT] has been widely used to deal with scalability issues. Basically, the access control is managed through capabilities. A capability, originally introduced at [cbac], could be seen as a ticket specifying access rights to subjects. If a subject does possess a ticket it has the proof of the holder rights to access the object.

In PRISTINE, we propose a new access control architecture combining both CBAC and ABAC assets. This architecture exploits RINA design features, in particular its layer management functions, by designing a generic solution able to secure any management layer function (routing or flow allocation for example). Furthermore, it leverages the adaptive and dynamic nature of ABAC model and the fine-grained authorization provided by the CBAC model.

4.3. Proposed CBAC architecture

4.3.1. Overview

The proposed access control architecture combines the advantages of ABAC model with those of CBAC model. On one hand, it is based on **the AC profiles** of the subject, the requested object, and the environment of the request. An AC profile refers to a set of attributes that are relevant to the authorisation decisions with respect to the studied scenario.

On the other hand, each subject is assigned an access **token** (or shortly, a token) that materializes its **capabilities** corresponding to its rights with respect to the system resources. These rights are basically derived from the AC profiles. The token is generated upon request originated from the subject. Once generated, the token is attached to any access operation request to check the permissions granted to this operation requester Tokens have a validity time; thus AC decision is based on dynamic profiles. Furthermore, the advantage of token usage is to avoid the evaluation of the AC profiles each time a request is issued (as in ABAC). This evaluation may

generate an overhead especially when these AC profiles are stored in an external system.

AC profiles are generic. They are instantiated based on the configured **AC policy**. For instance, we can imagine that any subject is characterized by a **group** and its resources have a specific **type** that make its access authorisation to a specific resource different from another subject from another group. We will see in Section 4.6 a scenario example.

To reinforce the system security (from token forgery for instance), the token content is **digitally signed** using the private key of the token issuer. Thus, receiving this token, the requestee is able to determine whether the token content has been tampered or not (see Section 4.3.4). Furthermore, tokens have durations of validity and are intended for specific audience.

In the remaining of this section, we first describe the different components of the proposed architecture and the token structure. Second, we explain how access control decision is made.

4.3.2. Architecture

For the sake of compatibility with other standardized AC architectures, we adopt the terminology commonly used in the domain (like in XACML [xacml] for instance). Furthermore, our design is generic enough to fit many use cases, mainly securing the layer management of RINA.

As depicted in Figure 7, the proposed CBAC architecture involves the following components:

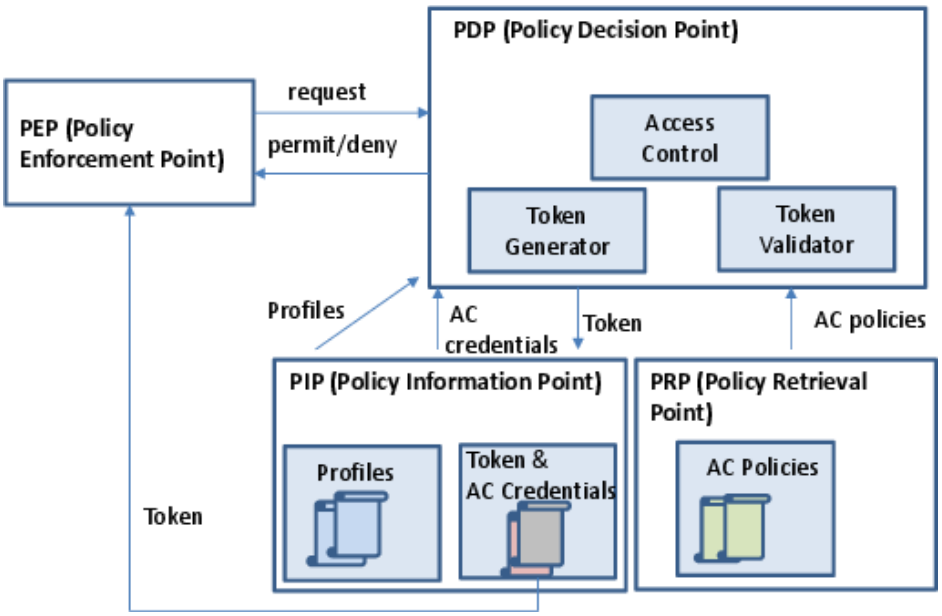

Figure 7. Proposed CBAC architecture.

- **The PEP (Policy Enforcement Point)** generates access requests and fill them with required information.
- **The PDP (Policy Decision Point)** evaluates the request based on the AC policy and returns an authorisation decision to the PEP. PDP is also in charge of token generation, as well as its checking and validation.
- **The PIP (Policy Information Point)** acts as an AC information storage. It stores i) the token used by the PEP, and ii) the AC credentials like the RSA keys and iii) the AC profiles used by the PDP.
- **The PRP (Policy Retrieval Point)** acts as a storage entity of the AC policies used by the PDP to make authorisation decisions.

Note that the PIP and the PRP can be a local database, an external database or even flat files.

4.3.3. Token Structure

Similarly to JSON Web Tokens (JWT) [json-rfc], the token includes the following information:

1. token_id: The unique identifier of the token.
2. issuer_id: The identifier of the issuer of the token.
3. holder_id: The identifier of the token holder.
4. issued_time: The time and date of the token generation.

5. audience: Identifies the recipients that the token is intended for; if the token is received by an entity which is not part of the audience, this entity ignores the request.
6. token_nbf: The time before which the token must not be accepted.
7. token_exp: The time after which the token must not be accepted.
8. cap: List of capabilities granted to the token holder, each one includes:
   - rType: The granted resource;
   - op: The granted operation over this resource.

To prevent token forgery, the token should be signed. Any request should include the token as well as its signature. More specifically, let T be the token content. The token issuer hashes the token using a predefined hash function H to obtain a hashed token $H(T)$. Then, it encrypts it using its private key, known uniquely by itself, to obtain the signature: $S = issuer\_pri\_key\_encrypt(H(T))$.

4.3.4. Token Generation and Access Control

We now explain the interaction and the flows exchanged between the different components of the CBAC architecture to ensure the token generation and the access control procedures.

Token Generation

The PEP generates a special request to get a token and sends it to the PDP. Depending on the use case, this request can be issued at any operation phase of the system, like at initialization for example. Upon reception of this request, the PDP decides if the subject is allowed to obtain a token (`Access Control' block). This decision is based on the AC profiles stored in the PIP and the AC policies stored at the PRP. Of course, these profiles and policies should be retrieved first to achieve this evaluation. If the decision is positive, the PDP generates the token (`Token Generator' block) and stores it in the PIP on the one hand, and returns it back to the PEP on the other hand.

Access Control

Whenever a subject wants to access a given system resource, the local PEP retrieves the user token from the PIP and sends it along with the request to

the peer PDP. Then, the first step the PDP performs, is the token validation by the 'Token validator' block (as explained below). Any required credential needed for this validation is obtained from the PIP. If the token is valid, the PDP checks if the requested resource appears in the token capabilities ('Access Control' block); in which case, the access is granted. Otherwise, the request is denied. In both cases, a reply is sent back to the peer PEP. The verification and validation of the token implies the check of the following items:

1. The audience: If the receiver is not in the token audience, it will discard the request.
2. The token owner: if the requester is not the owner of the token, the request is discarded.
3. The token validity time: the token cannot be accepted before its 'not before time' and after its 'expiration time'.
4. Signature: this step is crucial to ensure that the token was not tampered and that the signing entity is the real token issuer. For this, the receiver uses the plain token received *T* and the signature *S*. From one hand, it decrypts the signature using the public key of the token issuer (assumed to be known) to obtain: *issuer_pub_key_decrypt(S)*. From the other hand, it hashes the plain token to obtain *H(T)*. If both results are equal, then the token signature is valid. Otherwise, the token is ignored.

4.4. Access Control Scenarios in RINA

In a DIF, **IPCPs are considered as subjects** that are required to be authorised to proceed with some actions on the objects. Objects are the data stored in each IPCP RIB in the DIF. Basically, the CBAC policy will provide the corresponding capabilities to allow IPCPs to get access to these objects. Access Control in RINA is a crucial step that must be performed in various cases where different IPCPs would like to access to different resources. We consider two main scenarios: enrolment in a DIF and operation of layer management functions.

4.4.1. Enrolment Scenario

In RINA, enrollment is the process by which an IPCP communicates with another IPCP to join a DIF and acquires enough information to start

operating as a member of the DIF. After enrollment, the newly-enrolled IPCP is able to create and accept flows with other IPCPs within the DIF. As part of the procedure, the IPCP that is a DIF member may authenticate the joining IPCP via a specific policy. After authentication, the member IPCP must make an access control decision and allow the IPCP to join the DIF or reject its enrollment request.

4.4.2. Operation of Layer Management Functions Scenario

Layer management tasks operate through the exchange of CDAP messages invoking remote operations on the objects in peer IPCP RIBs. As introduced in the RINA overview, all incoming CDAP messages are processed by the RIB Daemon, which invokes an access control policy to check if the operation on the target object is allowed. Therefore, flexible and granular access control rules can be specified just in terms of object names and CDAP operations, regardless of the targeted layer management function. For example, routing could allow certain IPCPs with the right properties to perform read/write operations on the RIB objects that model the routing information state, but only allow read operations for other types of IPCPs. If needed, access control rules for individual objects and operations can be specified.

4.5. Integration and Specification of the Proposed CBAC architecture in RINA

This section describes the integration of the proposed CBAC architecture in RINA. Basically, the following updates on the RINA architecture are performed:

- CBAC is integrated in **the security management** component of each IPCP.
- **RINA DMS**: access control agents (ACA) are added to the RINA DMS management system at each system. Furthermore, AC Manager is added to the DMS Manager.

In the following, we detail these updates while considering the two aforementioned scenarios.

4.5.1. Updates on the RINA Management System for CBAC Integration

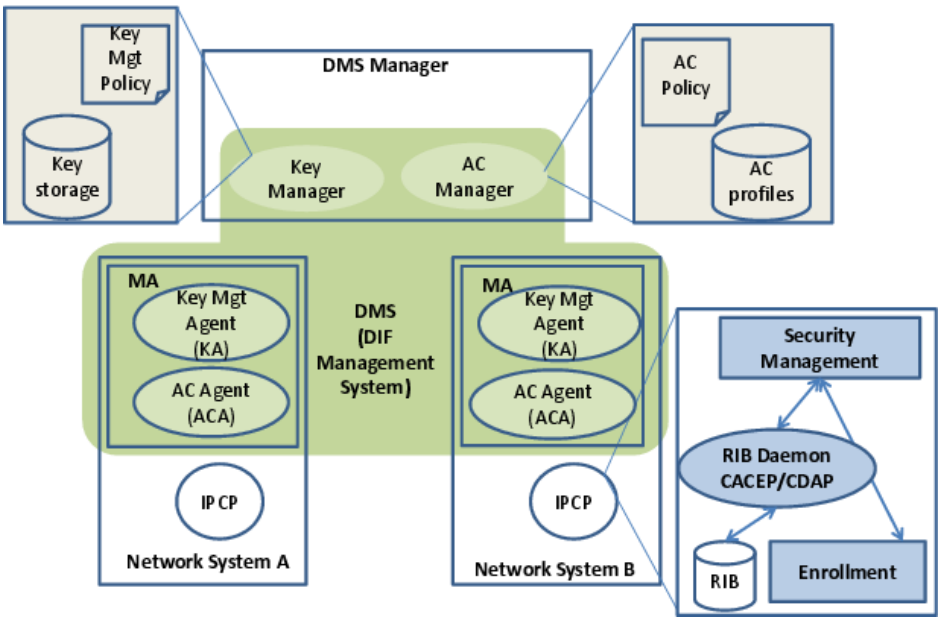

Figure 8. Updates on the RINA management system for CBAC integration

As illustrated by Figure 8, access control functions at any RINA network system is supported by the management functions involving the Management Agent (*MA*) of the system and the DMS Manager. Mainly, the following entities provide AC profiles and AC policies as the following:

1. At any system, two entities are involved: the Access Control management Agent (*ACA*) and the key management agent (*KA*). Both of them are part or sit along side with the *MA* of the system. The *ACA* distributes the AC profiles and AC policies to the IPCPs when needed while the *KA* provides the AC credentials (RSA keys, etc). Depending on the management system architecture, the IPCP can store this information locally or just make a request to get them when needed ([D4.2] and [D53]).

2. The *KA* and the *ACA* themselves are supplied with such information by the Key Manager and the AC Manager respectively. Both the Key Manager and the AC Manager are parts of the DMS Manager system. The Key Manager is responsible, among others, of generating and supplying different network systems of their DIFs with cryptographic

keys [1]. Similarly, the AC Manager stores the AC profiles and AC policies used inside the DIF.

Note that the specification of the interactions between the DMS Manager and the MAs are out of the scope of this section (see [D4.2] and [D53] for further details).

4.5.2. Access Control Scenarios in a RINA DIF with CBAC

To enable access control functions, the security management component of the RINA architecture is updated to integrate the CBAC architecture depicted in Figure 7 These updates are illustrated in Figure 9.

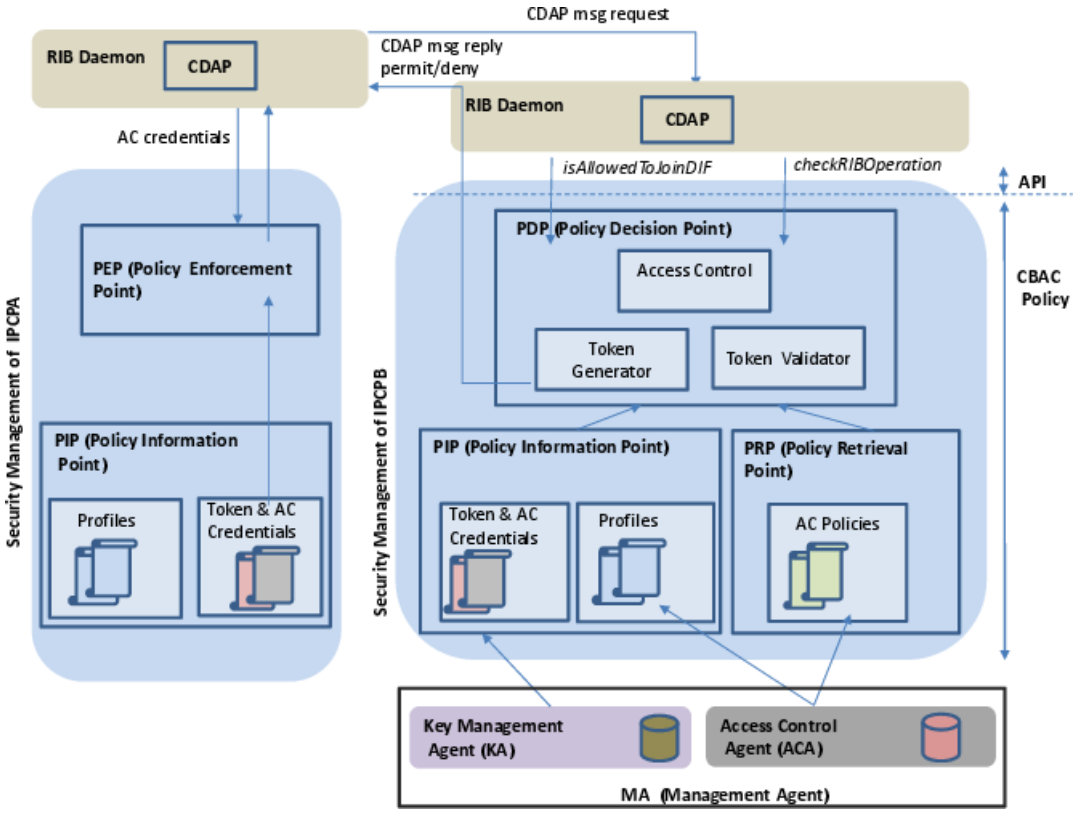

Figure 9. Access Control Scenarios in a RINA DIF with CBAC

Let us now consider two IPCPs: IPCP A and IPCP B and explain how token generation and access control is done between them (see Figure 9 [2]).

[1] For sake of simplicity, the presented blocks of the key manager are only a subset of its functions, see [D4.2] for further details.

[2] Note that some blocks of the IPCP A security management component are omitted for clarity reasons.

Enrollment Scenario

The token is generated following a request from IPCP A to IPCP B (precisely the enrollment request as explained hereafter). Receiving this request, the access control will be invoked by the RIB Daemon via the *isAllowedToJoinDIF* API operation. The PDP 'Access Control' block decides whether IPCP A is allowed to join the DIF. This decision is based on the AC policy and the AC profiles. Two cases are possible, either the AC profiles and AC policies are already present at the PIP and the PRP respectively, or these blocks should retrieve them from the *ACA*. In case a positive decision, the PDP 'Token Generator' block generates a token, stores it in the PIP and attaches it to the reply message.

Operation of layer management functions

Before sending a remote operation request, the RIB Daemon of IPCP A first looks for its token from the PIP. The token is attached to the CDAP message request. At IPCP B, this message is processed by the RIB Daemon which invokes the access control policy (via the *checkRIBOperation* API call). First step that the PDP performs is to check the token signature. For this, the cryptographic keys are obtained from the *KA* through the PIP. Finally, the 'Access Control' block of the PDP proceeds the operation request based on the token capabilities. In the following sections, we specify CBAC for both enrollment and operation of layer management functions scenarios.

4.5.3. Specification of the Enrollment Scenario

Figure 10 illustrates first steps of the enrollment triggered by IPCP A first establishes a connection with peer IPCP~B by sending a CDAP M_CONNECT message. After a successful authentication, the access control starts at IPCP B to check whether IPCP A is allowed to join the DIF. AC profiles are retrieved via the M_READ and M_READ_R CDAP messages with as parameter the name of the requested subject (AC profiles in this case). Then, based on this information, IPCP B makes the authorisation decision and generates the token. Obtaining the AC credentials is done via M_READ and M_READ_R CDAP messages.

Note that this information retrieval is not always necessary as information could be already present in the IPCP. It is the case, for instance, for RSA keys which are used in the authentication process prior to CBAC invocation.

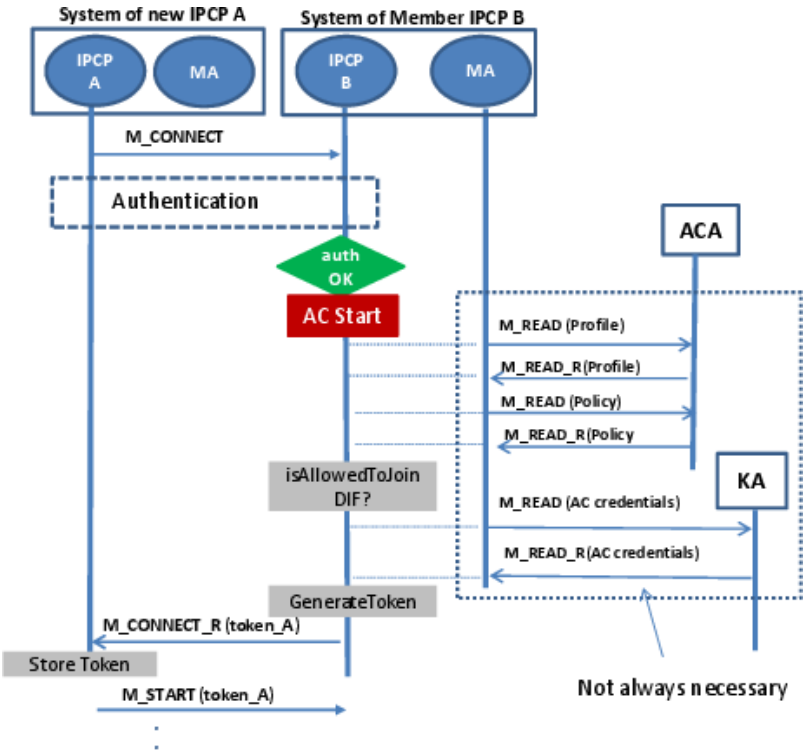

Figure 10. Specification of AC in enrollment scenario in RINA

To return the token to IPCP A, the M_CONNECT_R message is used. The received token is stored by IPCP A. At this level, any future operation performed by IPCP A will include its token, for instance the M_START operation. Detailing the remaining steps of the enrollment procedure is out of the scope of this paper. However, as they involve remote RIB operations, next section will provide hints on how access control of these operations is done. An important remark at this level is that this specification does not require new CDAP messages nor fundamental updates of the existing ones. This witnesses the advantages of the RINA design, in particular its separation of mechanisms and policies.

Specification of the Operation of Layer Management Functions Scenario

In Figure 11 an IPCP A wants to read an object from the RIB of a peer IPCP B. It sends a CDAP M_READ message. This message includes, in addition to the requested object, the token of the requestor. IPCP B first checks the validity of the token (as specified in Section 4.3.4). Keys necessary to perform this check are obtained from the *KA* via the M_READ and M_READ_R operations. After validating the request, IPCP B checks whether the operation is allowed, in which case, the M_READ_R message is sent with the requested object.

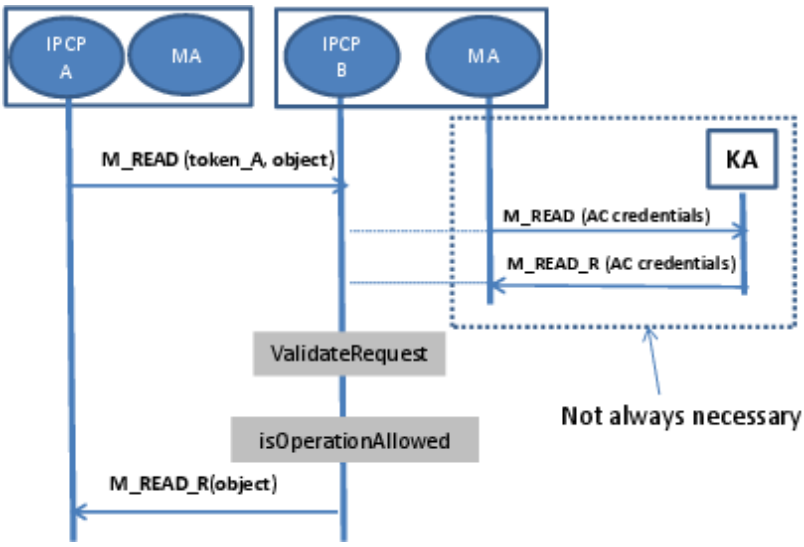

Figure 11. Specification of AC in operation of layer management functions scenario

4.6. Implementation of the CBAC Policy in the IRATI Stack

We have implemented a Proof of Concept (PoC) of the previously described CBAC policy in the IRATI stack.

4.6.1. CBAC Interaction with Other IRATI Components

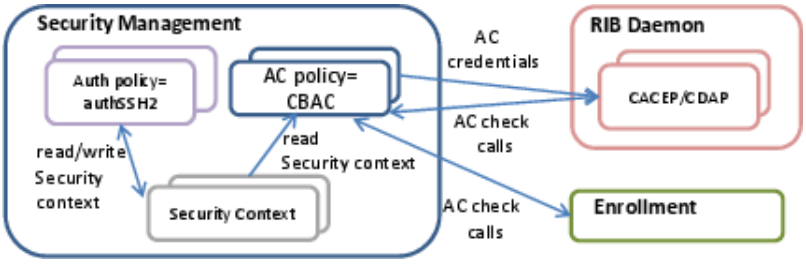

Figure 12. IPCP components that interact with the CBAC implementation

Figure 12 shows the integration of the CBAC policy in the IRATI stack.

CBAC is implemented as a policy in the security management component of the IPCP. Each time an access control is required, CBAC is invoked to make an authorisation decision. The security management component has in addition to CBAC, an authentication policy (e.g., SSH2). Both CBAC and authentication policy share a security context which stores the configuration of the authentication and data encryption policies. Thus, CBAC is configured with this security context (e.g., RSA keys). CBAC policy interacts with the RIB Daemon in two ways. First, before invoking any remote RIB operation, the RIB Daemon reads the token from the CBAC policy. Second, receiving any CDAP message, the RIB Daemon invokes

the CBAC policy to check whether the operation is permitted or not. Similarly, before starting the enrollment of a new member, the enrollment component needs to check whether the CBAC policy allows the new IPCP to join the DIF.

4.6.2. CBAC with Authentication Policy

With respect to the policy definitions, the aim of RINA is to make different policies fit together for a given scenario. Authentication and access control are a perfect example for this. Currently, there are three authentication policies specified in RINA (see [D4.2] for detailed information):

- *AuthNone*: The null case in which authentication is not required.
- *AuthNPassword*: The two IPCPs authenticate by proving they know a previously shared password.
- *AuthNAssymetricKey* (RSA): The two IPCPs 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.

In RINA, *AuthNone* and *AuthNPassword* policies are designed for scenarios assuming trusted environment. Thus, CBAC should always return a positive reply in these cases. Otherwise, when the DIF is configured to rely on the *AuthNAssymetricKey* policy, CBAC should perform the access control as described earlier in this document.

4.6.3. Technologies Used in CBAC Implementation

For cryptographic operations, CBAC relies on the openSSL libcrypto library [openssl], due to its widespread use and extensive feature set. In particular, the policy uses MD5 and SHA-256 hash functions, RSA key management, private encryption and public decryption. Both tokens and profiles are stored as JSON objects. In addition to the advantage that is a standardized representation, JSON provides an efficient method even in constrained environment such as Internet of Things [dcap]. Indeed, compared to other traditional formats such as XML, JSON provides a much finer grained control and a simpler data representation. As a consequence, requesting and processing time are considerably reduced.

4.6.4. Proof of Concept Scenario

Description of the Scenario

We have performed experiments with a PoC scenario to validate the CBAC integration with the rest of the IRATI implementation and to show a working example of the concepts introduced earlier in the paper. The experimental scenario is depicted in Figure 13.

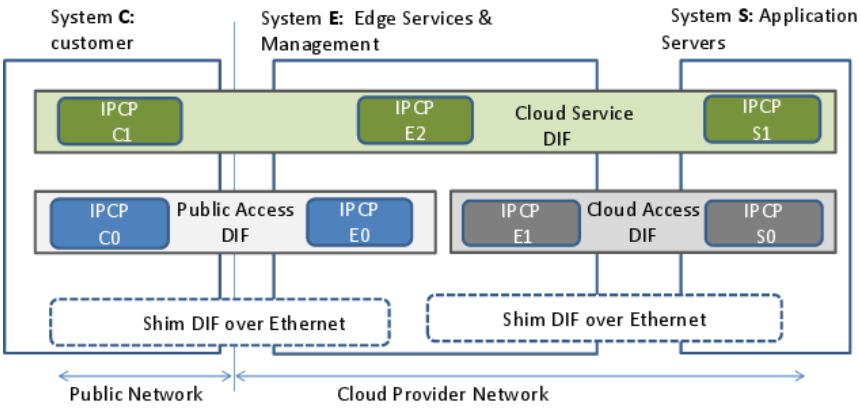

Figure 13. Experimental scenario

We consider three different systems, *a Customer* system (**C**) having access to cloud services provided by a cloud provider network. The latter includes an *Edge Services and Management system* (**E**) offering services to customers. The application running on system **E** is responsible of the delivery of content and advertisement of available services. The *Application Servers* system (**S**) runs an application offering data storage services. We define three DIFs: the *Cloud Service DIF* shared between the three systems. This DIF uses the services provided by the underlying *Public Access DIF* and *Cloud Access DIF*. Both *Public Access DIF* and *Cloud Access DIF* use the services provided by the shim DIF over Ethernet.

DIF Configuration

To enable CBAC integration, new parameters should be added to the DIF security management configuration.

1. "ACprofilestore": the storage file of the access control profiles (set to "AC-profile.json").
2. "ACPolicystore": the storage file of the access control policy (set to "AC-policy.json").
3. "DIFConfigstore": the DIF configuration file name.

4. "TokenGenIPCPName": the token generator IPCP name. In a DIF, the token generator has the privilege to generate its won token following the same algorithm as any other IPCP. Thus, the first time the RIB Daemon requests the token from the security management, the token generator generates its token and returns it to the RIB Daemon to be able to perform remote operations. Thus, the 'TokenGenIPCPName' configuration parameter is required.
5. "EncryptAlgo": the encryption algorithm used in token signature generation (set to "AES128" for instance)

As an example, the security management configuration of the DIF cloud-access.DIF ("cloud-access.dif" file) is:

```
...
"securityManagerConfiguration":{
   "policySet":{
      "name":"cbac",
      "version":"1",
      "parameters":[ {
            "name":"ACprofilestore",
            "value":"AC-profile.json"
         }, {
            "name":"TokenGenIPCPName",
            "value":"S0"
         }, {
            "name":"EncryptAlgo",
            "value":"AES128"
         }, {
            "name":"ACPolicystore",
            "value":"AC-policy.json"
         }, {
            "name":"DIFConfigstore",
            "value":"cloud-access.dif"
         }
      ]
   },
   "authSDUProtProfiles":{
      "default":{
         "authPolicy":{
            "name":"PSOC_authentication-ssh2",
            "version":"1",
            "parameters":[ {
                  "name":"keyExchangeAlg",
                  "value":"EDH"
```

```
                }, {
                   "name":"keystore",
                   "value":"private_key"
                }, {
                   "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"
          }
       }
    }
 },
 ...
```

Access Control Profiles

Running the IRATI stack with this specific scenario does not requires updates on the core CBAC implementation. What we have specified is the profiles of IPCPs, DIFs and RIBs as well as an access control policy. These steps can in practice be done by the network administrator. Again, this highlights the flexibility of the RINA architecture regarding the definition of granular management layer access control policies. This also shows how RINA design maximises specifications and developments reuse.

Table 1 summarises the profiles defined for our use case scenario.

Table 1. Access control profiles for the access control scenario

	Name	Group	Type
IPCPs	C0,C1	customer	customer
	E0, E1, E2	customer	edge
	S0, S1	customer	server
RIBs of C0, C1	flow	customer	private
	enrollment	customer	private
RIBs of E0, E1, E2	flow	customer	private
	enrollment	customer	private
RIBs of S0, S1	flow	customer	private
	enrollment	customer	private
DIF	Cloud service DIF	customer	management
	Public access DIF	customer	access
	Cloud access DIF	customer	access

Each IPCP, DIF or RIB is characterized by a *group* and a *type*. With respect to the network system, we have defined *customer, edge and server* types. Furthermore, as all systems are related to customer services (not enterprises as we could imagine), they belong to *customer* group.

Concerning the RIB, we present two examples of RIB objects: enrollment and flows (for RIB information related to enrollment and flows allocation respectively). To reinforce the security of the system, the type of these objects is private. Finally, the *Cloud Service DIF* is devoted to the *management* of the requested services, while the two others are *access* DIFs.

The Json AC profile file considered in this scenario (Ac-profile.json) is:

```
{
  "IPCPProfiles": [ {
     "apName": "S0",
     "ipcpType": "server",
     "ipcpGroup": "customer",
     "ipcpRibProfile": [ {
        "ribGroup": "flow",
        "ribType": "private"
        }, {
        "ribGroup": "neighbors",
        "ribType": "private"
        } ]
      }, {
      "apName": "E1",
      "ipcpType": "edge",
      "ipcpGroup": "customer",
      "ipcpRibProfile": [ {
         "ribGroup": "flow",
         "ribType": "private"
         }, {
         "ribGroup": "neighbors",
         "ribType": "private"
         } ]
       }, {
       "apName": "S1",
       "ipcpType": "server",
       "ipcpGroup": "customer",
       "ipcpRibProfile": [ {
          "ribGroup": "flow",
          "ribType": "private"
          }, {
          "ribGroup": "neighbors",
          "ribType": "private"
          } ]
        }, {
        "apName": "E2",
        "ipcpType": "edge",
        "ipcpGroup": "customer",
        "ipcpRibProfile": [ {
           "ribGroup": "flow",
           "ribType": "private"
           }, {
           "ribGroup": "neighbors",
           "ribType": "private"
```

```
            } ]
          } ],
      "DIFProfiles": [ {
         "difName": "cloud-access.DIF",
         "difType": "access",
         "difGroup": "customer"
         }, {
         "difName": "public-access.DIF",
         "difType": "access",
         "difGroup": "customer"
         }, {
         "difName": "cloud-service.DIF",
         "difType": "management",
         "difGroup": "customer"
      } ]
}
```

Access Control Policy

Concerning the AC policy, the rationale is that in order to have access to cloud services, customers can establish direct connections with **E** systems but not with **S** systems. More specifically, focusing on the enrollment AC policy, any IPCP should obey the following rules:

1. An enrollment request issued by an IPCP from a different group should be denied.
2. An IPCP with type *server* can only permit enrollment of IPCPs:
   a. of type *edge* or *server* within a DIF of type *access*.
   b. of type *edge*, *server* or *customer* within a DIF of type *management*.
3. *edge* IPCPs allow enrollment from *customer* or *edge* IPCPs.

Note that these AC rules take into consideration the DIF profile as well as the IPCP profiles. By applying these rules, the following enrollment operations are allowed: from **E1** to **S0**, from **C0** to **E0**, from **E2** to **S1** and finally from **C1** to **E2**.

According to the same rationale, tokens capabilities generated by enroller IPCPs include all operations over enrollment objects (like, for instance, reading a watchdog object supervising enroller state) and flow creation objects.

As mentioned above, the profiles and policies are stored in Json files.

The Json AC policy file used in this scenario (*AC-policy.json*) is:

```
{
   "enrollment":[  {
         "rule":[  {
               "ipcpType":"whatever",
               "difType":"primary",
               "memberIpcpType":"whatever",
               "memberDifType":"whatever"
            }
         ],
         "capabilities":[  {
               "ressource":"all",
               "op":"all"
            }
         ]
      }, {
         "rule":[  {
               "ipcpType":"primary",
               "difType":"whatever",
               "memberIpcpType":"whatever",
               "memberDifType":"whatever"
            }
         ],
         "capabilities":[  {
               "ressource":"all",
               "op":"all"
            }
         ]
      }, {
         "rule":[ {
               "ipcpType":"edge",
               "difType":"access",
               "memberIpcpType":"server",
               "memberDifType":"access"
            }, {
               "ipcpType":"server",
               "difType":"access",
               "memberIpcpType":"server",
               "memberDifType":"access"
            }, {
               "ipcpType":"server",
               "difType":"access",
               "memberIpcpType":"edge",
               "memberDifType":"access"
            }, {
```

```
            "ipcpType":"edge",
            "difType":"access",
            "memberIpcpType":"edge",
            "memberDifType":"access"
        }, {
            "ipcpType":"customer",
            "difType":"access",
            "memberIpcpType":"edge",
            "memberDifType":"access"
        }, {
            "ipcpType":"server",
            "difType":"management",
            "memberIpcpType":"edge",
            "memberDifType":"management"
        }, {
            "ipcpType":"edge",
            "difType":"management",
            "memberIpcpType":"customer",
            "memberDifType":"management"
        }
    ],
    "capabilities":[ {
            "resource":"/resalloc/fsos",
            "op":"4_M_CREATE"
        }, {
            "resource":"/difManagement/enrollment/watchdog",
            "op":"8_M_READ"
        }, {
            "resource":"/difManagement/enrollment",
            "op":"14_M_START"
        }, {
            "resource":"/difManagement/enrollment",
            "op":"16_M_STOP"
        }, {
            "resource":"/difManagement/nsm/dft",
            "op":"4_M_CREATE"
        }, {
            "resource":"/resalloc/fsos",
            "op":"12_M_WRITE"
        }, {
            "resource":"/fa/flows/key",
            "op":"4_M_CREATE"
        }, {
            "resource":"/fa/flows/key",
            "op":"6_M_DELETE"
        }, {
```

```
                "resource":"/difManagement/enrollment/neighbors",
                "op":"4_M_CREATE"
            }, {
                "resource":"/difManagement/nsm/dft",
                "op":"4_M_CREATE"
            }, {
                "resource":"/difManagement/opstatus",
                "op":"14_M_START"
            }, {
                "resource":"/difManagement/nsm/dft/key",
                "op":"6_M_DELETE"
            }
        ]
    }
  ]
}
```

This file contains the list of capabilities that should be granted to a requestor at the enrollment and the conditions (rules) under which these capabilities are granted. Indeed, receiving and enrollment request, the security management triggers a procedure to parse this file with as parameters: the profiles of the current DIF, the new DIF member and the DIF member. Whenever a rule matching these profiles is found (same DIF profile, same DIF member profile, same new DIF member profile), the enrollment is allowed and the capabilities following this rule are granted to the requestor. For instance, according to the first rule, a *primary* DIF type allows *all* operations over *all* resources independently from the other entities attributes (*whatever*). If no rule corresponding to the studied profiles is found, the enrollment request should be denied.

Prototype Validation

We implement this policy and configure the system with the predefined profiles and policies. The three systems in Figure 13 are VMs hosted in the same physical machine (i7 2.9GHz CPU, 32 GB of RAM and 500 GB hard disk).

We first show the log output of the enrollment procedures from **E1** to **S0** and from **E2** to **S1**. After launching the IPCM manager, we see that the IPCPs **E0**, **E1** and the Shim DIF over Ethernet have been created. On the *edge* system, we issue the enrollment commands one after the other.

```
rina@rina:/pristine/userspace/etc/scenario$ socat - UNIX:../../var/log/
ipcm-console.sock
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
 | Port-ids of flows provided)
    1 | test-eth-vlan:1:: | shim-eth-vlan | ASSIGNED TO DIF 110 | E1-1-- |
 -
    2 | test-eth-vlan2:1:: | shim-eth-vlan | ASSIGNED TO DIF 111 | E0-1--
 | -
    3 | E1:1:: | normal-ipc | ASSIGNED TO DIF cloud-access.DIF | E2-1-- |
 -
    4 | E0:1:: | normal-ipc | ASSIGNED TO DIF public-access.DIF | E2-1-- |
 -
    5 | E2:1:: | normal-ipc | ASSIGNED TO DIF cloud-service.DIF | - | -

IPCM >>> enroll-to-dif 3 cloud-access.DIF 110 S0 1
DIF enrollment successfully completed in 42 ms

IPCM >>> enroll-to-dif 5 cloud-service.DIF cloud-access.DIF S1 1
DIF enrollment successfully completed in 52 ms
```

Similarly, we issue enrollment commands between other IPCPs. On **C** system, we trigger enrollment of IPCP **C0** to IPCP **E0** and from IPCP **C1** to IPCP **E2**.

```
rina@rina:/pristine/userspace/etc/scenario$ sudo socat - UNIX:/pristine/
userspace/var/log/ipcm-console.sock
[sudo] password for rina:
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
 | Port-ids of flows provided)
    1 | test-eth-vlan2:1:: | shim-eth-vlan | ASSIGNED TO DIF 111 | C0-1--
 | -
    2 | C0:1:: | normal-ipc | ASSIGNED TO DIF public-access.DIF | C1-1-- |
 -
    3 | C1:1:: | normal-ipc | ASSIGNED TO DIF cloud-service.DIF | - | -

IPCM >>> enroll-to-dif 2 public-access.DIF 111 E0 1
DIF enrollment successfully completed in 37 ms

IPCM >>> enroll-to-dif 3 cloud-service.DIF public-access.DIF E2 1
DIF enrollment successfully completed in 42 ms

IPCM >>>
```

We see that the AC policy defined in the CBAC policy allow these enrollment operations. This enrollment time is essentially due to the cryptographic operations required by the CBAC policy. Recall that CBAC operations do not require to add new messages to the original messages defined in RINA, but only local operations, which reduces the time overhead.

Now, we run a 'rina.apps.echotime.server' on the server side (system **S**) and a 'rina.apps.echotime.client' application on the edge side (system **E**). On the server side, we see the following lines:

```
30472(1465915714)#librina.logs (DBG): New log level: INFO
30472(1465915714)#librina.nl-manager (INFO): Netlink socket connected to
 local port 30472
```

On the client side, we have the following output:

```
12369(1465835837)#librina.logs (DBG): New log level: INFO
12369(1465835837)#librina.nl-manager (INFO): Netlink socket connected to
 local port 12369
Flow allocation time = 8.715 ms
SDU size = 20, seq = 0, RTT = 0.65026 ms
SDU size = 20, seq = 1, RTT = 0.64518 ms
SDU size = 20, seq = 2, RTT = 0.63413 ms
SDU size = 20, seq = 3, RTT = 0.94987 ms
SDU size = 20, seq = 4, RTT = 0.64106 ms
SDU size = 20, seq = 5, RTT = 1.1435 ms
SDU size = 20, seq = 6, RTT = 1.4731 ms
SDU size = 20, seq = 7, RTT = 0.86902 ms
SDU size = 20, seq = 8, RTT = 0.94171 ms
SDU size = 20, seq = 9, RTT = 1.157 ms
SDUs sent: 10; SDUs received: 10; 0% SDU loss
Minimum RTT: 0.63413 ms; Maximum RTT: 1.4731 ms; Average RTT:0.91049 ms;
 Standard deviation: 0.28342 ms
SDUs sent: 0; SDUs received: 0; 0% SDU loss
Minimum RTT: 9.2234e+18 ms; Maximum RTT: 0 ms; Average RTT:0 ms; Standard
 deviation: -0 ms
```

4.7. Annex: AC threats and countermeasures

In the previous sections, a comprehensive policy for access control was proposed. In this section, we discuss the different threats that any attacker

can apply over this policy and propose some countermeasures to avoid them. We consider both scenarios: enrollment and operation of layer management functions scenarios.

The threats are classified based on the subject of the attack.

4.7.1. Enrollment scenario

Let us consider two IPCPs: IPCP A which wants to join a DIF by sending an enrollment request to IPCP B. As previously explained, the AC profiles are retrieved from the *ACA*. In this scenario, the *ACA*, the IPCP A and the IPCP B can run the following threats:

Threats that the *ACA* runs

1. Threat1: Disclosure of the AC profiles: a non trusted IPCP could request the profiles from the *ACA*.
2. Threat 2: Eavesdropping profiles
3. Impact: As a result of these threats, a malicious IPCP can guess the token and uses it to impersonate the IPCP A.
4. Countermeasures:
    a. We suppose that the *ACA* is trusted.
    b. The IPCP requesting the profiles should be authenticated.
    c. Profiles should be digitally signed.
    d. The requests should be encrypted.

Threats that the IPCP B runs

1. Threat1: Receive wrong profiles following a man in the middle attack that can elevate or reduce the capabilities of the Profile's owner.
2. Impact1: the generated token is not coherent with the real profiles.
3. Threat2: Malicious IPCP obtains the token destined to IPCP A
4. Threat3: Eavesdropping tokens after listening the transmissions between the IPCP A and IPCP B.
5. Countermeasures:
    a. Encryption of requests
    b. Authentication of IPCP A

c. Signing profiles

Threats that the IPCP A runs

1. Threat1: IPCP A may receive an altered token. This can happen after profiles modification, if the IPCP B is malicious and deliver altered tokens, or after a man in the middle attack between IPCP A and IPCP B.
2. Countermeasures:
   a. IPCP B should be trusted
   b. Token should be digitally signed

4.7.2. Scenario 2: RIB Access

In this scenario, we suppose that there is an IPCP A which wants to access a resource from the RIB of a remote IPCP B.

Threats that the IPCP A runs:

1. Threat1: Eavesdropping token
2. Countermeasures: Encryption of requests

Threats that the IPCP B runs

1. Threat1: A malicious IPCP impersonates IPCP A
2. Threat2: Replay the token
3. Threat3: Abuse of the token capabilities by its owner (this IPCP adds some privileges to its token)
4. Threat4: IPCP A generates/guesses the token
5. Threat6: Replay requests between A and B
6. Threat7: Eavesdropping the communication between IPCP A and IPCP B
7. Countermeasures:
   a. Authentication of IPCP A
   b. Digital signature of the token
   c. Reducing validity time of tokens
   d. Encryption of data

e. Reducing the validity time of the keys (to lower the probability to guess the keys and decrypt the data)

5. Multi-Level Security

Multi-Level Security (MLS) refers to access control mechanisms for protecting data or "objects" that can be classified at various sensitivity levels, from processes or "subjects" who may be cleared at various levels of trust. 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, but typically makes it difficult to share data effectively. 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.

MLS solutions for the current dominant networking architecture, Internet Protocol (IP), are well understood and developed. However, these are unsatisfactory in terms of security and assurance on the one hand, and data sharing and flexibility on the other. RINA offers the potential for MLS solutions with greater security and flexibility than is possible with IP, but how to provide MLS in these architectures as not been considered so far.

5.1. MLS in RINA

When implementing MLS in RINA, separating data at different levels can be achieved using SDU protection, which is a module in the IPC Process that applies protection to outgoing SDUs according to a configured policy. For example, a policy that cryptographically protects the confidentiality and integrity of an SDU before it is relayed over an underlying DIF. This ensures that the data cannot be inappropriately read from the communication channel and that data at different classification levels is not inappropriately mixed.

However, 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 cannot be achieved using SDU protection and needs to be

carefully controlled to prevent accidental or deliberate release of sensitive information by users or malicious code.

In RINA, there is no mechanism to enable the secure transfer of data between classification levels in an automated way. The only means of sharing data is via manual transfer, i.e., where a person to checks the true classification level of the data to be transferred, and re-enters 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.

Traditional networks, e.g., those based on TCP/IP, use Boundary Protection Components (BPCs) to provide automated control of data flows between security levels. There are three main methods of automated boundary protection (i.e. without a human in the loop) used to prevent accidental or deliberate release of sensitive information: label checking, e.g. Purple Penelope [Gollmann] and DeepSecure XML Guard [DeepSec]; deep content inspection, e.g. Nexor Watchman [Nexor]; content modification, e.g. QinetiQ Sybard ® ICA Guard [Sybard]. However, current BPCs will not operate over RINA in a way that is non-bypassable and transparent to applications.

This section specifies a means of integrating the functionality of a Boundary Protection Component (BPC) into a RINA-based network. The RINA BPC enables data to be sent between security levels in a carefully controlled and secure way to prevent accidental or deliberate release of sensitive data. For example, it may be used to control flows of data between security levels, 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.

The RINA BPC is dictated by policy, meaning it can operate in dynamic environments that are facing frequent changes and unpredictable threats, where we may need to rapidly respond to external events by just modifying the MLS policies.

5.2. BPC Phases

Here we propose methods for achieving the functionality of a BPC in RINA. The RINA BPC intercepts data packets that are sent between different security levels and inspects the data. If the data is not suitable for the

recipient, then boundary protection is enforced, which may involved blocking or modifying the data. This functionality controls the flow of data between security levels, e.g. from a High system to a Low system, to ensure that data transferred from the High system is actually at a suitable classification level for the Low system. It may also be used to control data imported to the sensitive network, e.g. to check for malware.

There are three phases of the BPC: initialisation, operation and monitoring.

5.2.1. Initialisation Phase

The Initialisation Phase configures the network prior to operation. The aim of this phase is to ensure that IPCPs can only enrol in DIFs that are appropriate for their clearance level and that all data flows between different security levels are routed via the BPC.

During the initiation phase, the Distributed Network Management System (DMS) creates and configures the network. The central Network Domain Manager instructs the Management Agent (MA) on each node to create all the IPCPs that are needed in the network. The MA is the focal point for communication of the node with a Network Domain Manager. The MA on each system then configures the policies of each IPCP in the machine and assigns them to DIFs. It also provides the IPCPs with any credentials needed to enrol in their assigned DIFs. The policies and credentials are stored in each IPCP's Resource Information Base (RIB). The IPCPs then enrol in a DIF that is appropriate for their clearance level, using the configured credentials to authenticate to an existing DIF member in another machine.

The MA in the BPC node creates the BPC processes, which may be Application Processes (AP) or IPCPs, in a trusted node that operates at two or more classification levels. The BPC processes are provided with credentials and enrols in the DIFs or DAFs needed to form a bridge between nodes at different classification levels.

Once the processes have joined DIFs or DAFs, the Network Domain Manager and Management Agents configure the routing policies to ensure that all routes between security levels are via the BPC.

5.2.2. Operational Phase

The Operational Phase follows the Initialisation Phase. The aim of this phase is to control application data sent between two networks at different

security levels to ensure it is appropriate for the recipient. The data sent from an AP is routed via the BPC node according to the routing policy that was configured during the Initialisation Phase. The BPC then inspects the data packets; makes a decision whether the packet can be forwarded to the recipient; and enforces the decision. If the BPC determines that the data is not appropriate for the destination, it takes appropriate action according to its policy.

There are several options how the BPC handles SDUs that are unsuitable for the destination, depending on the threat model and types of data being inspected. The BPC may decide to block the packet from being forwarded. Alternatively, it may modify the packet before forwarding it, e.g. redacting the packet to remove sensitive data. It may extract the payload from the packet and create a new packet containing the payload from the original packet, performing a protocol break. This can mitigate attacks on the protocol.

5.2.3. Monitoring Phase

The Monitoring Phase runs in parallel with the Operational Phase. The aim of this phase is to ensure that the BPC is operating correctly; to detect malicious behaviour and to ensure that the BPC is not adversely affecting the performance of the network.

Monitoring is performed by the DMS. The Manager configures the MA on the BPC node to monitor specific parameters and to send a notification when the parameters exceed a threshold. The Manager, through the Management Agent, can modify the BPC's policies at any time according to the immediate needs. This allows the Manager to actively manage the actions of the BPC process in response to some indicator, minimising the risk of any information leakage.

5.3. BPC Specification and Design

The BPC requires two policies: PDU Intercept policy and the Boundary Protection policy. The PDU Intercept Policy intercepts all PDUs as they cross the boundary between networks at different classifications, regardless of their intended destination. It then passes the intercepted PDUs to the Boundary Protection policy to be inspected. The Boundary Protection policy may inspect the PDU header and/or SDUs, depending on the

architecture option, and makes a policy-based decision whether it is suitable for the intended destination. If the data is appropriate, the PDU is forwarded as normal. If the data is not appropriate for the recipient, the decision is enforced by taking appropriate action as specified in the Boundary Protection policy.

There are several architecture options for achieving a BPC in RINA, depending on the requirements of the scenario in which the BPC is to be deployed. The BPC functionality can be implemented at the application level, as in Option 1, at the DIF-level, as in Option 3, or it can be split between the application level and the DIF-level, as in Option 2. The placement and functionality of the BPC policies depends on the BPC architecture option.

5.3.1. BPC Architecture Option 1: BPC at an AP

Option 1 is to implement the BPC at the application-level with the sending application explicitly sending the data to the BPC for inspection. This operates in a similar way to a web proxy server, i.e. the application sends the data to the BPC, which inspects it and may forward it on to the intended destination.

Option 1 is shown in Figure 14. Here the BPC is a device sitting between a High and Low system that has a RINA Application Process in the DAF (BPC-AP), an IPCP connected to the High System DIF (IPCP-2) and an IPCP connected to the Low System DIF (IPCP-3). The BPC-AP implements the Boundary Protection policy. It is the endpoint of the flow with the sending application. This means that RMT in the N-level IPCP will deliver the PDU to a local EFCP instance, which then delivers it to the BPC-AP. Therefore the PDU Intercept Policy is not needed and no modifications are required to RINA. In addition, this means that the BPC-AP receives SDUs that have already been extracted from the PDUs and delimited by the SDU Delimiting module in the underlying IPCP, so it does not need to delimit SDUs.

A High application (AP-1) sends data to a Low application (AP-2). Since data is transferred through the N-level DIF, it is not possible to control the routing at the DAF-level. Therefore, to ensure that all data sent between High and Low APs flows through the BPC node, the network should be configured so that the all systems apart from the BPC only contain IPCPs

and APs at a single level and that IPCPs are only able to enrol in a DIF that is appropriate for their clearance level. This means that, during the initialisation phase, strong authentication in the N-level DIF must be used during enrolment. In addition, the BPC-AP must bridge both the High DAF and Low DAF, so that only the BPC node can relay data from one classification level to another. It therefore has two flows allocated: one provided by the High DIF to AP-1 and another provided by the Low DIF to AP-2. Routing policies must also be configured in the DIFs that ensure all routes between classification levels are via the BPC.

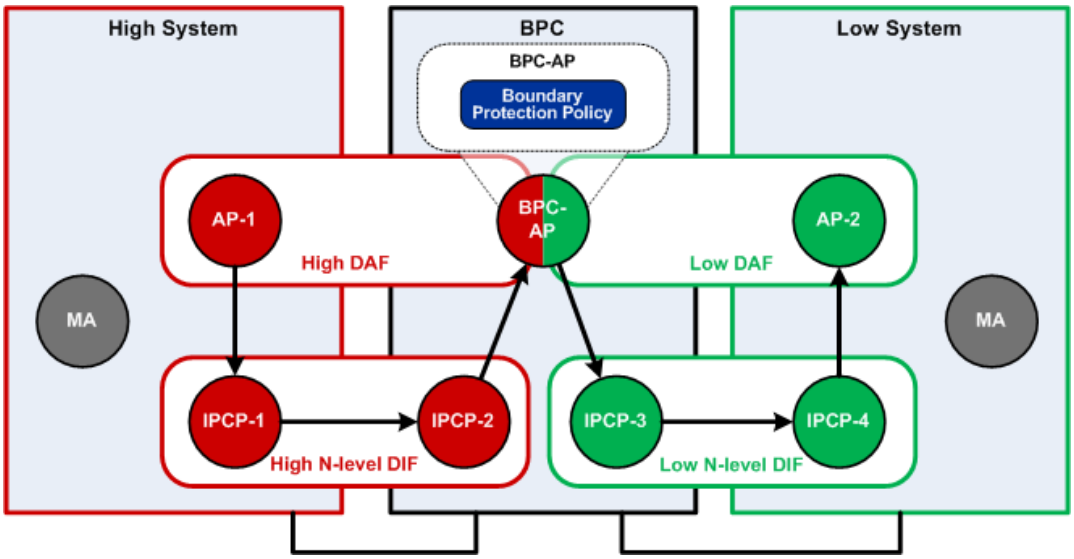

Figure 14. BPC at the Application Level

During the operation phase, AP-1 sends its SDUs to the BPC-AP and includes the intended recipient (AP-2) in the SDU itself.

The BPC-AP extracts the intended destination from the SDU and applies its Boundary Protection policy to the content of the SDU. If the data is appropriate, the BPC-AP sends the SDU to the intended destination (i.e. AP-2)), writing it to the flow provided by the Low DIF. Otherwise, the BPC-AP will enforce boundary protection, by taking action as defined in its Boundary Protection policy. This may include blocking the SDU or it modifying it, e.g. redacting sensitive content, before sending to the intended destination.

This approach allows the application data to be inspected to determine, through some knowledge of the data semantics, what its classification level is and/or that it does not contain hidden data. This BPC Option 1 has the advantage that it does not require modifications to RINA and RINA does not need to be aware of the security levels, as the inspection occurs in

an application process. However, it is not transparent to the applications, as they need to be modified to send data to the BPC-AP, rather than to the intended destination. In addition, it breaks the end-to-end connection between the two APs into two parts: from the AP-1 to the BPC-AP and from the BPC to AP-2. This means that if AP-1 uses encryption when sending data, AP-1 will negotiate a session key with the BPC-AP, enabling the BPC to decrypt the data before inspecting it. The BPC-AP will then use a separate session key, negotiated with AP-2, to re-encrypt the data before forwarding it to AP-2. It also breaks the end-to-end flow and error control, which may result in data loss due to congestion that can't be recovered.

5.3.2. BPC Architecture Option 2: BPC split between an AP and an IPCP

A second approach to implementing the BPC splits the BPC functionality between an application process in the DAF (BPC-AP) and an IPCP in the N-level DIF (BPC-IPCP). The BPC-AP is configured with a Boundary Protection policy and the BPC-IPCP is configured with a PDU Intercept policy, as shown in Figure 15. All PDUs transferred across the boundary between networks are intercepted by the PDU Intercept policy before being relayed, regardless of their intended destination, and passed to the BPC-AP to be inspected. The sending application addresses the PDU to the destination application, rather than to the BPC-AP. This approach avoids breaking the end-to-end connection between the sending and receiving applications by forcing the flow up to the DAF at the BPC node.

The initiation phase is as before, with strong authentication in the (N-1)-level DIFs to ensure that all IPCPs are only able to enrol in a DIF that is appropriate for their clearance level and a routing policy that ensures all routes between classification levels are via the BPC node.

During the operation phase, AP-1 sends its SDUs directly to AP-2, i.e. addressing the PDU to AP-2, using its underlying DIF; it does not send them explicitly to the BPC. Since the APs have different clearance levels, the data is routed via the BPC node according to the routing policy configured in the initialisation phase. At the BPC node, the packet is intercepted by the PDU Intercept policy at the BPC-IPCP and passed up to the BPC-AP. If the Boundary Protection policy is configured to inspect only the PDU header, then the BPC-AP applies the Boundary Protection policy to the PDU. If the Boundary Protection policy is configured to inspect the

SDU, then the BPC-AP delimits SDUs, i.e. by applying concatenation or segmentation, to extract the SDUs before applying the policy. It may also need to buffer SDUs. Once the SDU has been extracted, the BPC-AP then applies configured Boundary Protection policy to the SDU to ensure that it is appropriate for the PDU destination, i.e. that it does not contain sensitive or malicious content. If the content is appropriate for the destination, the PDU is forwarded as normal. If the data is not appropriate for the recipient, the BPC-AP will enforce the decision by taking appropriate action as specified in its policy. This may involve blocking the PDU from being forwarded or modifying the PDU, e.g. to remove sensitive content, before forwarding it to the destination, and subsequently updating the logging information.

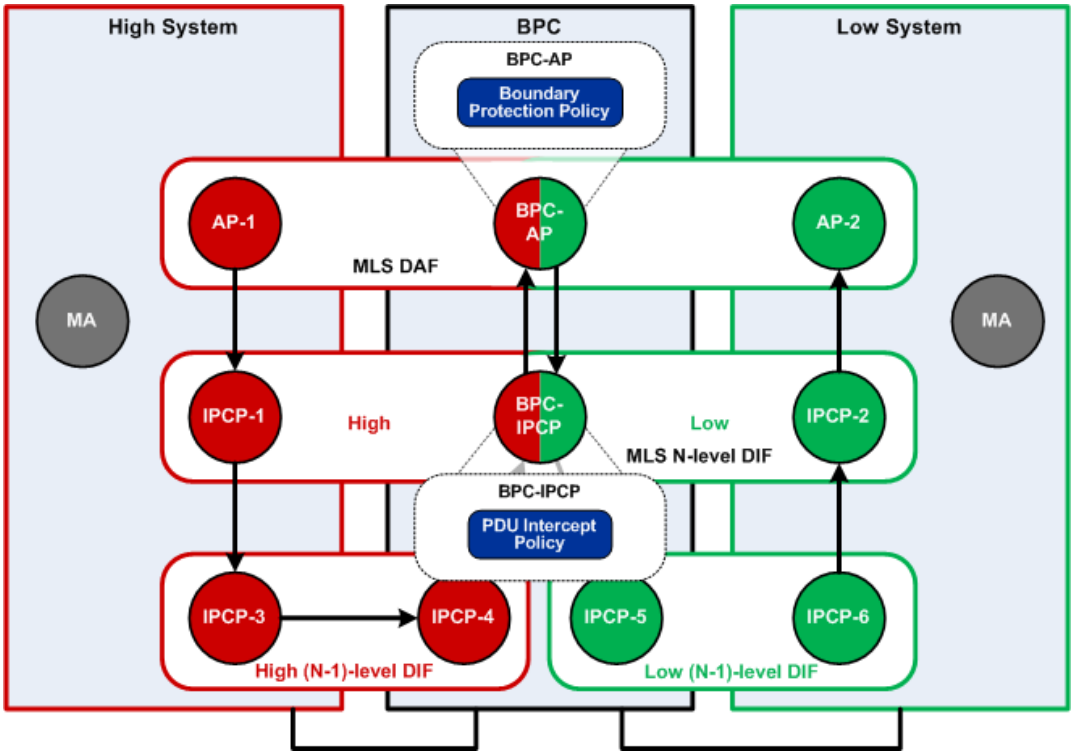

Figure 15. BPC split between AP and IPCP

This approach has the advantage that it is transparent to the applications, as the data is intercepted at the DIF-level and passed up to the application. This allows applications to be used unmodified. However, since the end-to-end connection between AP-1 and AP-2 is maintained, if encryption is used at the DAF-level, then the BPC-AP will need access to the session key to be able decrypt the data before performing the inspection. This option does not have the support of RINA model and may require some modifications to be implemented. It will also add some delay to the transmission of PDUs, as it introduces additional processing on them before forwarding.

5.3.3. BPC Architecture Option 3: BPC at an IPCP

An alternative means of implementing a BPC is to implement it at the relaying DIF-level, which means it is transparent to applications. The BPC node contains a modified IPCP (BPC-IPCP) that is configured with both a PDU Intercept policy and a Boundary Protection policy, as shown in Figure 16. All PDUs transferred across the boundary between networks are intercepted by the PDU Intercept policy before they are relayed, regardless of their intended destination. The BPC-IPCP then inspects the PDU and applies its Boundary Protection policy to decide whether it is suitable for the intended destination.

This option could be seen as an equivalent to a network level firewall. It is not application aware, as it does not have application context, so it cannot inspect application data (SDUs). The Boundary Protection policy is therefore configured to inspect only the PDU header or PCI. However, the default PCI in PDUs cannot be relied on for security based filtering (e.g., addresses are changed dynamically), so security labels are needed on the PDUs. IPCPs need to be trusted to apply the correct labels, which much be be cryptographically attached to the PDU so they cannot be modified or removed. These labels could be generated by trusted hardware or software when the PDUs are created in a similar way to having a data cryptor at every terminal.

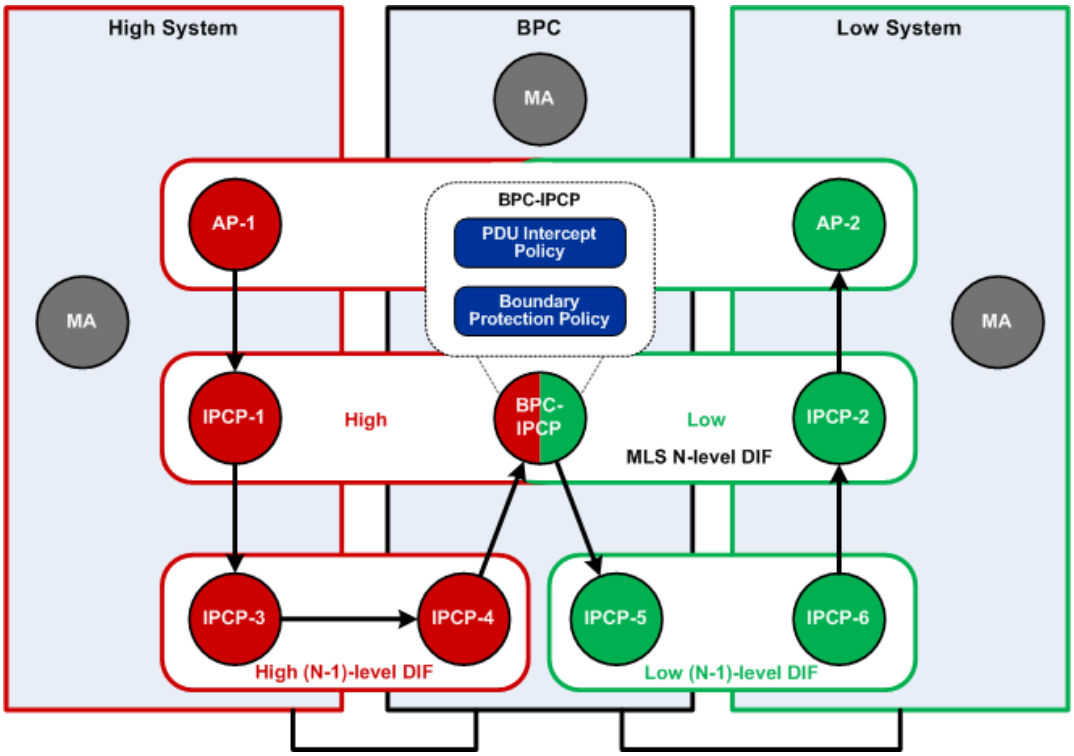

Figure 16. BPC at the DIF-level

During the initialisation phase, authentication is used to ensure that IPCPs in the (N-1)-level DIF used are only able to enrol in a DIF that is appropriate for their clearance level. The IPCPs are also configured with routing policies that ensure that all routes between security levels are via the BPC.

During the operation phase, AP-1 sends its PDUs directly to AP-2; it doesn't send them to the BPC. AP-1 passes the PDU to IPCP-1 in the underlying DIF, which in turn passes the PDU to IPCP-3 in the High (N-1)-level DIF. IPCP-3 routes the PDU to IPCP-4 in the BPC node, which passes it up to the BPC–IPCP, where it is intercepted by the PDU Intercept policy. The BPC-IPCP then inspects the PDU and applies its Boundary Protection policy. The Boundary Protection policy first cryptographically verifies that the label has not been modified or removed, e.g. by checking a signature or message authentication code. If the label is verified, the Boundary Protection policy is applied to the label. If the label is appropriate for the destination, the PDU is forwarded as normal. Otherwise, the BPC-IPCP will enforce the decision by taking appropriate action as specified in its Boundary Protection policy. This may involve blocking the PDU from being forwarded and subsequently updating the logging information. Since the label is checked at the network boundary, the sender AP does not need to know the security level of the recipient AP.

Since this option is not application aware, it cannot inspect application data (SDUs), which means it cannot be used to protect the trusted network from malware. It is therefore only suitable for protecting the high network from releasing sensitive data. It may require modifications to RINA PDU formats to enable the security labels to be applied.

5.4. Implementation of BPC

The Operation Phase of Option 1 has been implemented using the PRISTINE SDK. The Initialisation Phase is achieved by manually configuring the IPC Manager on each node using the IPC Manager configuration file. The monitoring phase can be achieved using the DMS Manager and Management Agent being developed in WP5 and so is not described here.

The BPC-AP is implemented as a Java application using the Librina Java library to send and receive data over RINA. It is built using Apache Maven [Maven] and packaged as a Java Archive (JAR) file. It uses the Spring Framework [Spring] for dependency injection and configuration of the application.

The BPC-AP has one application entity (AE) for each network that it is protecting and each AE operates over a different DIF. For example, if the BPC is to protect traffic flows between a High network and a Low network, it will have two AEs: one that can operate over a High DIF and allocate flows to applications classified at 'High' and another that can operate over a Low DIF and allocate flows to applications classified at 'Low'.

5.4.1. SDU Format

The BPC expects to receive SDUs containing a BPC Request with the intended destination of the message and the data to be inspected.

The BPC Request contains the following fields:

- ApplicationProcessNamingInformation destination: stores the application naming information of the intended destination application process,
- OBJECT data: stores the data to be sent to the intended destination

Where ApplicationProcessNamingInformation contains the following fields:

- STRING processName: name of the process.
- STRING processInstance: instance of the process.
- STRING entityName: name of the entity.
- STRING entityInstance: instance of the entity.

5.4.2. Boundary Protection Policy

The implemented Boundary Protection Policy has three components:

- Data Security Level Classifier: determines the security level of the data
- Application Security Level Classifier: determines the security level of the intended destination application
- Rule Engine: uses rules to determine whether the data is at a suitable level for the intended destination

Each component has a defined interface, allowing with multiple implementations of each component. This means that the BPC application can be reconfigured without compilation using the Spring Framework's dependency injection.

The component interfaces and implementations are described below.

Data Security Level Classifier

The Data Security Level Classifier interface has the following function:

- **getClassification**: This takes as input the data to be classified and returns the Security Level, which is an enum with a value if either HIGH or LOW.

A Keyword Data Security Level Classifier has been implemented that searches the data for sensitive keywords. If any of the sensitive keywords are present, then the data is classified as HIGH, otherwise it is classified as LOW.

Application Security Level Classifier

The Application Security Level Classifier interface has the following function:

- **getClassification**: This takes as input the Application Process Naming Information of both the application and the DIF over which the application operates and returns the Security Level, which is an enum with a value if either HIGH or LOW.

An Application Security Level Classifier has been implemented that determines the classification of an application from the name of its DIF. If the DIF name contains a sensitive keyword, it is classified as HIGH, otherwise it is classified as LOW. This is used to classify both the sending application and the intended destination application.

Rule Engine

The Rule Engine interface has the following function:

- **makeDecision**: This takes as input the security level of the sender, the security level of the intended destination and the security level of the data. It returns a Rule Decision, which contains a Decision enum of ACCEPT, REJECT or INDETERMINATE and a Reason string value containing the reason for the decision.

A Rule Engine has been implemented with the rules listed in Table 2.

Table 2. Boundary Protection Policy Rules

ID	Rule Summary	Data Classif.	Sender Classif.	Dest. Classif.	Decision	Reason
1	Any classification can send or receive LOW data	LOW	HIGH or LOW	HIGH or LOW	ACCEPT	Data is classified as LOW
2	LOW applications cannot send HIGH data	HIGH	LOW	HIGH or LOW	INDETER-MINATE	Data classification exceeds sender's clearance.
3	HIGH applications can	HIGH	HIGH	HIGH	ACCEPT	Destination is cleared to

ID	Rule Summary	Data Classif.	Sender Classif.	Dest. Classif.	Decision	Reason
	send and receive HIGH data					receive HIGH data
4	LOW applications cannot receive HIGH data	HIGH	HIGH	LOW	REJECT	Destination is not cleared to receive HIGH data

ACCEPT means that the SDU is at a suitable classification for the destination. REJECT means that the classification of the SDU is above that of the destination. INDETERMINATE means that the classification of the sender is lower than that of the SDU. This should not occur, since the sender should not have data that is above its classification level.

The Rule Decision is enforced by the BPC application. If the decision is ACCEPT, then the SDU is sent to the intended destination application. If the decision is either REJECT or INDETERMINATE, the SDU is not forwarded. The BPC application records both the decision and action taken in its log file.

5.5. Verification of BPC Function

Verification is regarded as the process of checking whether the proposed BPC solution complies with the design specification in order to function correctly as expected. To verify the BPC implementation, two additional Java applications has been implemented: a sending application and a receiving application. The applications are configured with a security classification of either High or Low. The sending application is configured to send at SDUs either classified at 'High' or 'Low' to the BPC for forwarding to the receiving application.

5.5.1. Experimentation Environment

The experimentation environment used for the verification tests is shown in Figure 17. It consists of three VMs running the PRISTINE SDK, which are connected via two VLANs: 110 and 111. Nodes 1 and 2 are configured to have a classification level of either High or Low, depending on the requirements of the verification test.

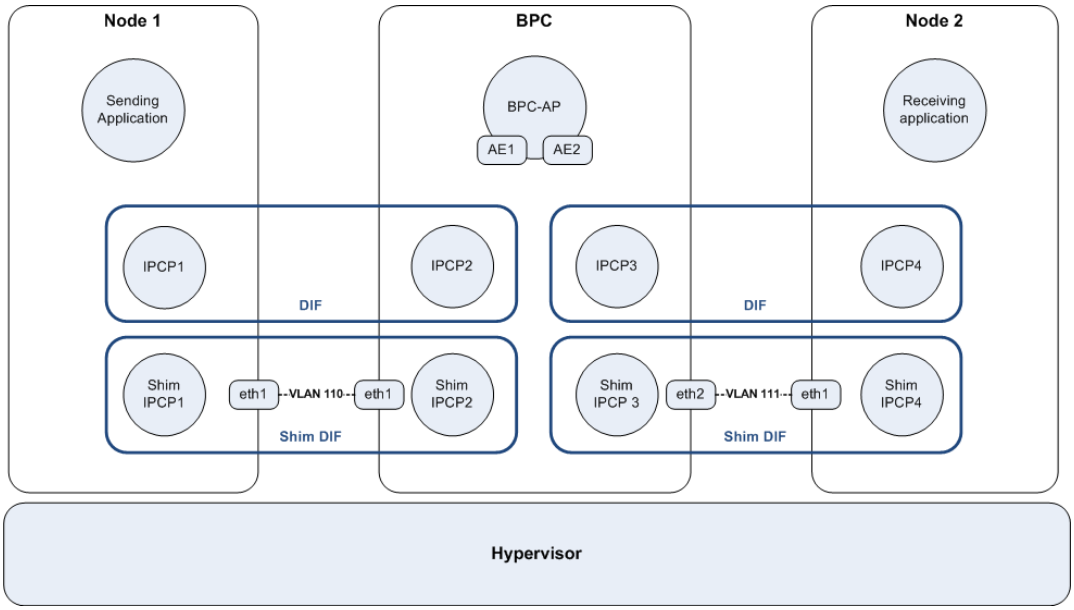

Figure 17. Experimentation Environment

Verification Test Set-up for Tests 1 and 2

Verification Tests 1 and 2 require one High node and one Low node. The Experimentation Environment for Tests 1 and 2 is shown in Figure 18.

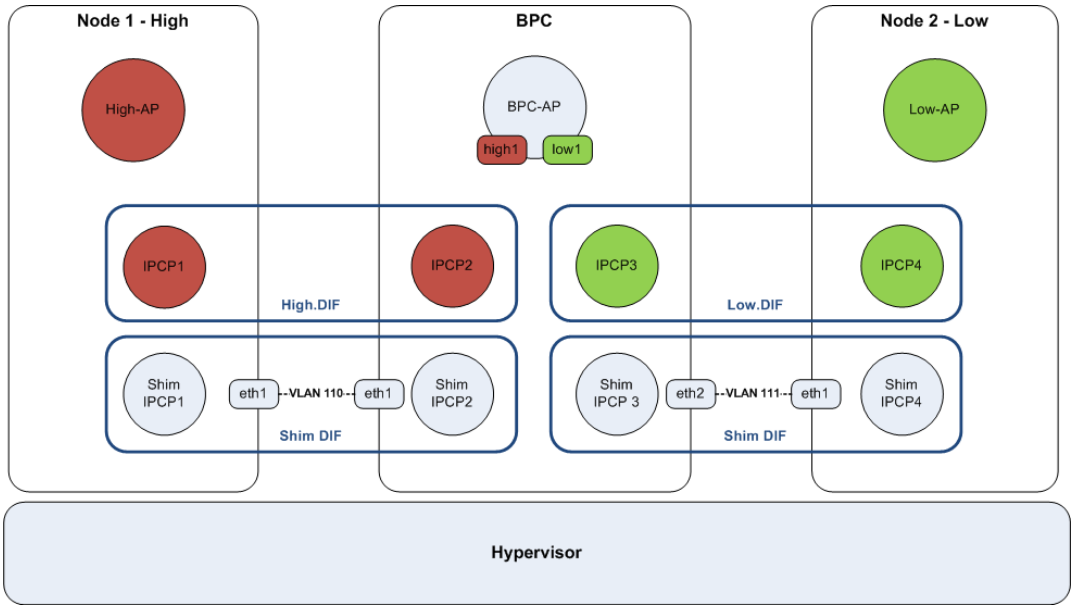

Figure 18. Experimentation Environment for Verification Tests 1 and 2

Node 1 is configured as a node classified at 'High'. It runs the High application (High-AP) and has a normal IPCP, IPCP1, classified at High that is enrolled in the High DIF, as well as a Shim IPCP (Shim-IPCP1) than runs over VLAN 110. For Test 1, High-AP is the sending application and Low-AP is the receiving application. For Test 2, Low-AP is the sending application and High-AP is the receiving application.

The second VM is configured as the BPC. It runs the BPC application and has two normal IPCPs (IPCP2 and IPCP3) and two Shim IPCPs (Shim-IPCP2 and Shim-IPCP3). IPCP2 is classified at High and is enrolled in the High DIF, while IPCP3 is classified at Low and is enrolled in the Low DIF. Shim-IPCP2 runs over VLAN 110 and Shim-IPCP3 runs over VLAN 111.

The third VM is configured as a node classified at 'Low'. It runs the Low receiving application (Low-AP) and has a normal IPCP, IPCP4, classified at Low that is enrolled in the Low DIF. It also has a Shim IPCP (Shim-IPCP4) than runs over VLAN 111.

The BPC-AP has AEs (High1 and Low1) each with an allocated flow: High1 has a flow to High-AP using the High DIF and Low1 has a flow to Low-AP using the Low DIF.

Verification Test Set-up for Test 3

Verification Test 3 requires two High nodes. The Experimentation Environment for Test 3 is shown in Figure 19.

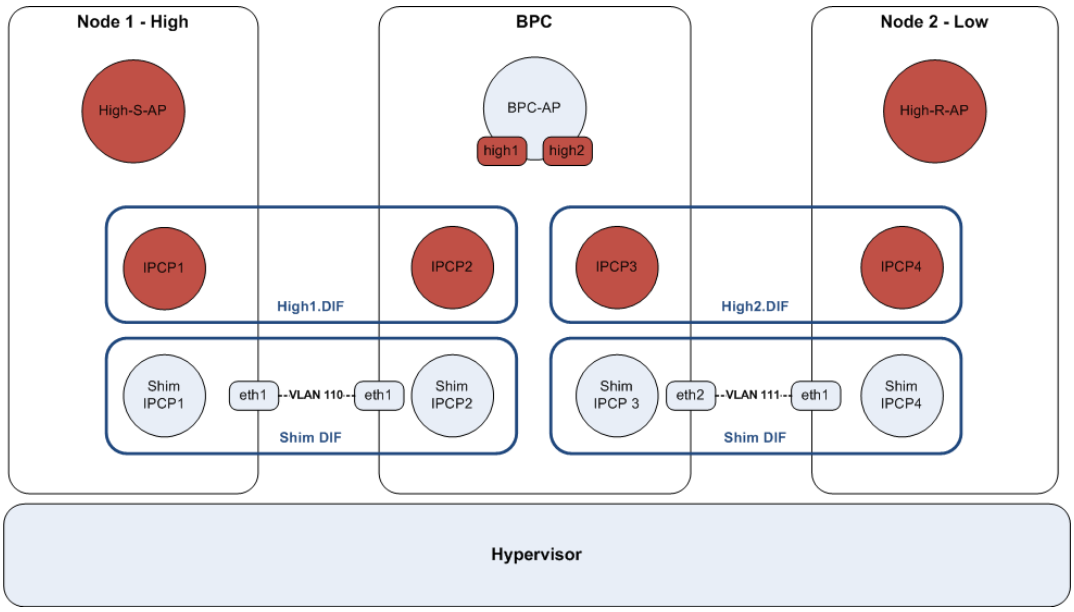

Figure 19. Experimentation Environment for Verification Test 3

Node 1 is configured as a node classified at 'High'. It runs the High sending application (High-S-AP) and has a normal IPCP, IPCP1, classified at High that is enrolled in the High1 DIF, as well as a Shim IPCP (Shim-IPCP1) than runs over VLAN 110.

The second VM is configured as the BPC. It runs the BPC application and has two normal IPCPs classified at High (IPCP2 and IPCP3) and two Shim

IPCPs (Shim-IPCP2 and Shim-IPCP3). IPCP2 is enrolled in the High1 DIF, while IPCP3 is enrolled in the High2 DIF. Shim-IPCP2 runs over VLAN 110 and Shim-IPCP3 runs over VLAN 111.

The third VM is configured as a node classified at 'High'. It runs the High receiving application (High-R-AP) and has a normal IPCP, IPCP4, classified at High that is enrolled in the High2 DIF. It also has a Shim IPCP (Shim-IPCP4) than runs over VLAN 111.

The BPC-AP has AEs (High1 and High2) each with an allocated flow: High1 has a flow to High-S-AP using the High1 DIF and High2 has a flow to High-R-AP using the High2 DIF.

Verification Test Set-up for Test 4

Verification Test 4 requires two Low nodes. The Experimentation Environment for Test 4 is shown in Figure 20.

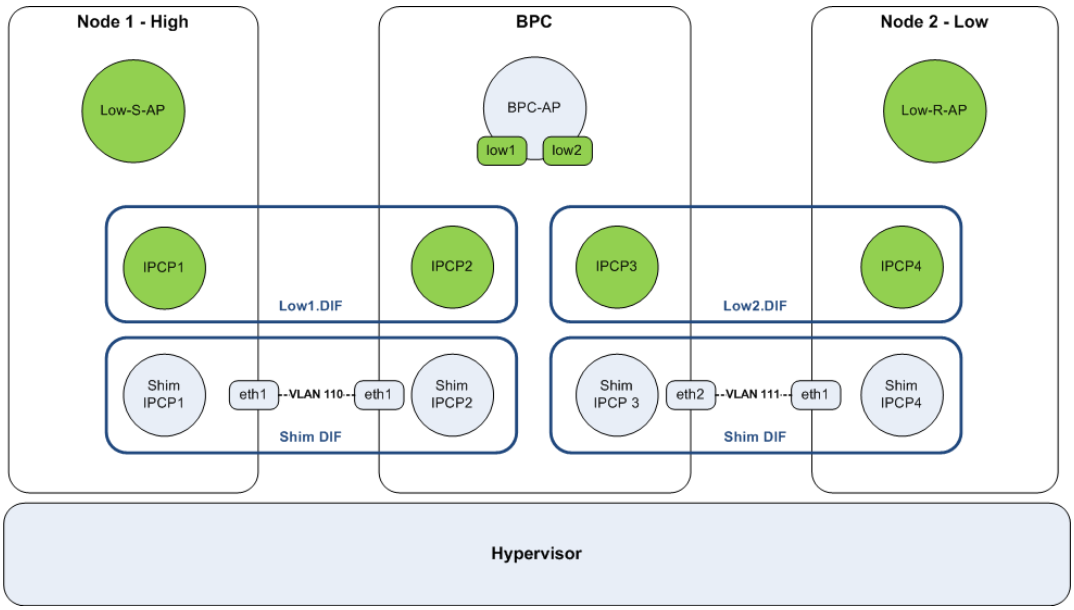

Figure 20. Experimentation Environment for Verification Test 4

Node 1 is configured as a node classified at 'Low'. It runs the Low sending application (Low-S-AP) and has a normal IPCP, IPCP1, classified at Low that is enrolled in the Low1 DIF, as well as a Shim IPCP (Shim-IPCP1) than runs over VLAN 110.

The second VM is configured as the BPC. It runs the BPC application and has two normal IPCPs classified at Low (IPCP2 and IPCP3) and two Shim IPCPs (Shim-IPCP2 and Shim-IPCP3). IPCP2 is enrolled in the Low1 DIF,

while IPCP3 is enrolled in the Low2 DIF. Shim-IPCP2 runs over VLAN 110 and Shim-IPCP3 runs over VLAN 111.

The third VM is configured as a node classified at 'Low'. It runs the Low receiving application (Low-R-AP) and has a normal IPCP, IPCP4, classified at Low that is enrolled in the Low2 DIF. It also has a Shim IPCP (Shim-IPCP4) than runs over VLAN 111.

The BPC-AP has AEs (Low1 and Low2) each with an allocated flow: Low1 has a flow to Low-S-AP using the Low1 DIF and Low2 has a flow to Low-R-AP using the Low2 DIF.

5.5.2. Verification Tests and the Results

Test 1: Data Flow from High to Low

Test Summary: This test is to verify the functionality of the BPC when an application with a classification of High sends data to an application with a classification of Low. The aim of the test is to verify that the BPC blocks the Low application from receiving SDUs classified at High, i.e. Rule 4 in Table 2.

Initial Conditions: The High application is configured to send 10 SDUs to the BPC application to be forwarded to the Low application. The first 5 SDUs contain data classified at High, while the remaining SDUs contain data classified at Low.

Checks to be performed in the test: The High and Low applications record the number of SDUs sent and received in their log files, while the BPC records the decision for each SDU. At the end of the test the log files from all three applications are inspected to determine the classification level of the sent SDUs, the number of SDUs blocked by the BPC and the number of SDUs received at the destination application. This information is then used to determine whether the BPC made the correct decision and enforced the decision.

Expected Results: The BPC should block all SDUs classified at High and forward all SDUs classified at Low to the Low-AP. The Low-AP should receive all of the Low SDUs.

Results: The test has been conducted using the experimentation environment shown in Figure 18. The results of the test are shown in Table 3.

Table 3. Results of verification test 1

SDU number	SDU classification	BPC Decision	SDU Received at destination
1	High	REJECT	No
2	High	REJECT	No
3	High	REJECT	No
4	High	REJECT	No
5	High	REJECT	No
6	Low	ACCEPT	Yes
7	Low	ACCEPT	Yes
8	Low	ACCEPT	Yes
9	Low	ACCEPT	Yes
10	Low	ACCEPT	Yes

The results in Table 3 show that the BPC blocked all of the High SDUs and forwarded all of the Low SDUs. It therefore operates as expected when a High application sends SDUs to a Low application.

Test 2: Data Flow from Low to High

Test Summary: This test is to verify the functionality of the BPC when an application with a classification of Low sends data to an application with a classification of High. The aim of the test is to verify that the BPC allows the Low application to send SDUs classified at Low to the High application (Rule 1 in Table 2), but prevents it sending SDUs classified at High (Rule 2).

Initial Conditions: The Low application is configured to send 10 SDUs to the BPC application to be forwarded to the High application. The first 5 SDUs contain data classified at Low, while the remaining SDUs contain data classified at High.

Checks to be performed in the test: The Low and High applications record the number of SDUs sent and received in their log files, while the BPC records the decision for each SDU. At the end of the test the log files from all three applications are inspected to determine the classification level of the sent SDUs, the number of SDUs blocked by the BPC and the number of

SDUs received at the destination application. This information is then used to determine whether the BPC made the correct decision and enforced the decision.

Expected Results: The BPC should allow the LOW SDUs to be forwarded to the High-AP, but block the HIGH SDUs. The High-AP should receive all of the SDUs.

Results: The test has been conducted using the experimentation environment shown in Figure 18. The results of the test are shown in Table 4.

Table 4. Results of verification test 2

SDU number	SDU classification	BPC Decision	Received at destination
1	Low	ACCEPT	Yes
2	Low	ACCEPT	Yes
3	Low	ACCEPT	Yes
4	Low	ACCEPT	Yes
5	Low	ACCEPT	Yes
6	High	INDETERMINATE	No
7	High	INDETERMINATE	No
8	High	INDETERMINATE	No
9	High	INDETERMINATE	No
10	High	INDETERMINATE	No

The results in Table 4 show that the BPC allowed all of the SDUs to be forwarded to the High application. It therefore operates as expected when a Low application sends SDUs to a High application.

Test 3: Data Flow from High to High

Test Summary: This test is to verify the functionality of the BPC when an application with a classification of High sends data to another application with a classification of High. The aim of the test is to verify that the BPC allows the destination application to receive SDUs classified at both Low and High, i.e. Rules 1 and 3 in Table 2.

Initial Conditions: The High sending application is configured to send 10 SDUs to the BPC application to be forwarded to the High receiving

application. The first 5 SDUs contain data classified at High, while the remaining SDUs contain data classified at Low.

Checks to be performed in the test: The High applications record the number of SDUs sent and received in their log files, while the BPC records the decision for each SDU. At the end of the test the log files from all three applications are inspected to determine the classification level of the sent SDUs, the number of SDUs blocked by the BPC and the number of SDUs received at the destination application. This information is then used to determine whether the BPC made the correct decision and enforced the decision.

Expected Results: The BPC should allow all SDUs and forward them all to the destination. The High-R-AP should receive all of the SDUs.

Results: The test has been conducted using the experimentation environment shown in Figure 19. The results of the test are shown in Table 4.

Table 5. Results of verification test 3

SDU number	SDU classification	BPC Decision	Received at destination
1	High	ACCEPT	Yes
2	High	ACCEPT	Yes
3	High	ACCEPT	Yes
4	High	ACCEPT	Yes
5	High	ACCEPT	Yes
6	Low	ACCEPT	Yes
7	Low	ACCEPT	Yes
8	Low	ACCEPT	Yes
9	Low	ACCEPT	Yes
10	Low	ACCEPT	Yes

The results in Table 5 show that the BPC allowed all of the SDUs and forwarded them all. It therefore operates as expected when a High application sends SDUs to another High application.

Test 4: Data Flow from Low to Low

Test Summary: This test is to verify the functionality of the BPC when an application with a classification of Low sends data to another application

with a classification of Low. The aim of the test is to verify that the BPC allows the destination application to receive SDUs classified at Low, but blocks SDUs classified at High, i.e. Rules 1 and 2 in Table 2.

Initial Conditions: The Low sending application is configured to send 10 SDUs to the BPC application to be forwarded to the Low receiving application. The first 5 SDUs contain data classified at High, while the remaining SDUs contain data classified at Low.

Checks to be performed in the test: The Low applications record the number of SDUs sent and received in their log files, while the BPC records the decision for each SDU. At the end of the test the log files from all three applications are inspected to determine the classification level of the sent SDUs, the number of SDUs blocked by the BPC and the number of SDUs received at the destination application. This information is then used to determine whether the BPC made the correct decision and enforced the decision.

Expected Results: The BPC should forward all Low SDUs and block all High SDUs. The Low-R-AP should receive only the Low SDUs.

Results: The test has been conducted using the experimentation environment shown in Figure 20. The results of the test are shown in Table 6.

Table 6. Results of verification test 4

SDU number	SDU classification	BPC Decision	Received at destination
1	High	INDETERMINATE	No
2	High	INDETERMINATE	No
3	High	INDETERMINATE	No
4	High	INDETERMINATE	No
5	High	INDETERMINATE	No
6	Low	ACCEPT	Yes
7	Low	ACCEPT	Yes
8	Low	ACCEPT	Yes
9	Low	ACCEPT	Yes
10	Low	ACCEPT	Yes

The results in Table 6 show that the BPC accepted the High SDUs and forwarded them to the Low receiving application. The decision for the

Low SDUs was INDETERMINATE and the SDUs were not forwarded. The therefore test verified that the BPC operates as expected when a Low application sends SDUs to another Low application.

6. Key Management System: specification

6.1. Considerations for Key Management

The Key Management Function is a component of the Management DAF, used to create and manage the keys assigned to the IPC Processes (any key material required for the Management Agents of managed systems to join the NMS-DAF must be previously distributed by other means). The Key Management Function provides a boundary between the quotidian activities of the RINA environment and the curation of key material [3], the authorised use of such key material in security functions, and the curation of and access to suitably strong entropy. In PRISTINE, the Central Key Manager is part of the Central Manager DAP. Each Management Agent includes a Key Management Agent function that communicates with the Central Key Manager, using a private encrypted protocol that is opaque to the rest of the system.

Each key has a life-cycle: it is created, lives a useful life, and is retired [4]. The length of a key's life-cycle and what exactly happens between its creation and retirement can vary widely depending on the specific application.

The issue of managing key material needs to be considered in a broader context than any individual protocol or subsystem, since the security of the whole system is limited by its weakest component. This leads to three cases for RINA key management, that we will refer to as 'multi-tenant', 'enterprise' and 'standalone'

In the multi-tenant case, there are a number of independent Central Key Managers, corresponding to different security domains. This would apply, for example, in a datacentre, whose overall RINA network will be co-ordinated by a central manager, but different tenants will want to retain control over the management of their key material.

In the enterprise case, RINA key management is part of a more general key management, in which case the assumption is that there will be a DAF

[3] Key material is that data access to which is required to be able to perform security related activities.

[4] [NIST] describes this more fully in §7. Life-cycle names used in this document are taken from this NIST publication.

with a small number of distinguished instances acting as a protocol gateway between the RINA requirements and a KMIP [KMIP] installation.

In the standalone case there will be:

1. some local information that is accessible by the IPCP without any active network connectivity and
2. given successful enrolment with the management DIF, access to an appropriate subset of key material.

In PRISTINE we will not consider the enterprise case in detail.

In all these scenarios, the principal function of the key management system is to deny access to key material without appropriate authorisation.

6.1.1. Time-of-day related issues

In order to constrain the time available to an attacker to break an individual key, a variety of systems (for example X.509 certificates) include a time after which the 'key' becomes invalid. This presupposes that the processes using this key have access to a reliable time-of-day source. Accurate time-of-day clocks may not be available on low-cost devices such as IoT nodes, however, so we need to give due consideration to issues of time in relation to key material.

When accurate and reliable local clocks are not available we must depend on time provided by the network, using protocols such as NTP, in order to establish time-of-day. This opens the possibility of an attack through spoofing or denial-of-service of the network time source. Having control of time on a node allows replay of old keys (by 'winding back time') or a security DoS attack (by pushing time forward so that keys used by the node become invalid).

Bounding the validity of key material based on time-of-day protects it from unbounded computational attack. This leads to the conclusion that the key material whose validity is dependent on the time-of-day (and date) should either be suspended (i.e. not used) if a node cannot be confident about the accuracy of its local clock (this may be the default state after power-on), or time-of-day constraints are ignored (as a matter of policy choice, depending on the application scenario; each choice opens different

attack vectors). However, in either case the local key management agent must not destroy keys simply because they are apparently invalid due to the current time, but should only do so on appropriately authenticated instruction from the key management master.

6.1.2. Legal intercept, validation and conformance testing

Depending on the jurisdiction and operational conditions, it may be necessary to supply key matter to enable decryption by authorised third parties [5].

There are two categories of access that may be required:

First category: post-hoc requirement to provide ephemeral keys so that intercepted transmissions can be decrypted (non real-time, may be time bounded by months/years) – note that this facility is also necessary in order to debug protocol issues that could otherwise become intractable due to encryption;

Second category: real-time 'eavesdropping', achieved by either supplying keys or the unencrypted stream, in a way that is not detectable by the end user – note that this is also needed to enable conformance testing.

To support the first category, we must ensure that the Key Manager archives keys without destroying them, even though Key Management agents may destroy their local copies. To support the second category, the Key Manager must be able to look up a session key on the basis of the identifier of an active association.

Implementing these capabilities is beyond the scope of PRISTINE; however the key management architecture must not preclude their being added later.

6.1.3. Bootstrapping and rebooting

Securely joining the management DIF requires some initial token to be provided. However it is essential for the key management system to be able to update this token (although there may be other channels available to do this, e.g. TR-069), so that subsequent (re)joining is not dependent on the

[5] Note that there is no known legal requirement for the ability to 'masquerade' as someone else.

original token (e.g. IoT nodes use a simple token to initially join that can then be updated to become more secure).

Given that accurate time-of-day may not be available in a bootstrap condition, and the policy choice offered in Section 6.1.1 joining the management DAF should be achievable by means of non-time-of-day-dependent keys.

6.1.4. Interworking between domains

While in PRISTINE the Key Management system belongs to the unitary management DAF, it should be possible to establish secure DIFs to a separate management domain. This has implications for the distribution of keys, for example that there is access to shared key repositories.

6.1.5. Bulk encryption

Ongoing bulk encryption of data streams is a quotidian activity of RINA layers that must be performed without direct involvement of the Key Manager. However the process of establishing the (usually symmetric) keys to perform the en/decryption will involve the Key Manager (if only to support the intercept, validation and conformance testing requirements above).

6.2. Key management architecture

The Key Management function has two components: the central Key Manager (which is a component of the Manager process); and Key Manager Agents (which are components of the Management Agents). These two components communicate via a subset of the Key Management Interoperability Protocol [KMIP] encapsulated in CDAP messages in the management DIF (the details of this encapsulation are for further study). The Key Management Agent provides part of the API of the local Management Agent to enable key matter-related operations. Keeping key material in a separate module is in accordance with best practice as described in §6.2 of [NIST].

Assumptions:

1. Key material is immutable (once created/written, it never changes - though it may be destroyed)

2. Key meta-data (e.g. validity) is held in RIB, can be changed
3. Key management components are stateless
4. Each KMA will be initialised with the necessary key material to enable its management agent to join the management DIF

The architecture is illustrated below in Figure 21. This shows that the Central Key Manager (CKM) communicates with the central manager, and each Key Management Agent (KMA) communicates with the corresponding Management Agent, in each case using a non-blocking message passing interface.

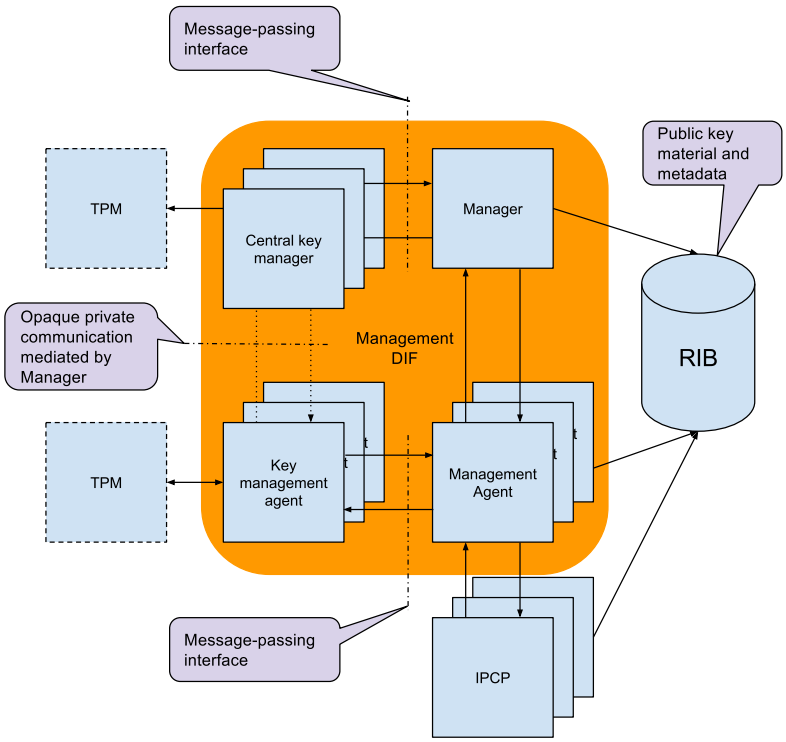

Figure 21. key management function architecture

6.2.1. Communication between the CKM and the KMAs

The Central Key Manager and the Key Management Agents communicate indirectly across the management DIF by exchanging data blocks via the Management Agents and Central Manager. This channel relies on the management DIF to provide authentication. However the possibility that the management DAF might be compromised means that this communication must be private, so the CKM and KMAs establish a shared secret via a Diffie-Hellman key exchange and use this to encrypt the data blocks they exchange to prevent long-term subversion and capture

of key material. They are capable of migrating to a new shared secret without loss of service (the shared secret may be expired due to time and/or volume of use parameters stored in the RIB). Since 'man in the middle' attacks are possible on this channel, the protocol will assume that replay attacks and message corruption may occur. However there will be no mitigation for erasure or load based attacks since denial of service is always unpreventable. There will be no assumption of in-sequence delivery, which allows multiple requests to be in flight for performance/scalability.

Key manager/agent communication is hierarchical, i.e. there is no communication between agents. Using indirection and recursion (but not referral) allows layers of hierarchy to be added for scalability. The system will allow for separate key managers so that different security domains can be established over the same network, supporting Enterprise and Multi-Tenant uses of the system.

6.2.2. Relationship to RIB

Information regarding the state and purpose of key material shall be held in the RIB. In particular, the Manager RIB shall be used to hold information about the availability and validity of key material, thus enabling automatic notification of all relevant entities. Public key material may also be held in the RIB, but private key material shall not be held in the RIB.

6.2.3. Storage of key material

Storage of key material presents an attack vector against the system. Since RINA deployments may extend into nodes whose levels of physical security will vary, the architecture must support restricting the distribution of key material according to the level of trust that can be placed in individual nodes.

The industry has developed Trusted Platform Modules (TPMs) to manage aspects of this problem. It is thus important for the architecture to allow key material to be stored in a TPM. Not every node may be equipped with a TPM. Nodes without a TPM may either support appropriately secure storage of key material within the OS kernel, or exploit the secure connection between the local Key Management Agent and the central Key Manager that has access to a TPM, so that the former can act as a proxy for the latter.

In order to provide the key material to enable the initial connection to the management DAF, there should be appropriate storage for at least the information needed to establish the management DAF. This could be a TPM providing secure storage, or some other device unlocked by a password etc. (e.g. SIM), or for low-end devices such as an IoT object some NVRAM with such data on it (which might not be secure under the ability to power-cycle and physically access the node).

Any centralised key store (e.g. RINA Key Manager and KMIP master) should be physically secure and have access to an un-compromised time-of-day clock.

6.2.4. Stateless operation

The Key Management agent shall be stateless with respect to the operation of the policies that use it. This may require the same parameters to be passed across the boundary more than once, but simplifies error recovery. Note that this implies that the contents of the Key Management Agents can always be reconstituted, thus avoiding the risks associated with local write-down paths due to backups.

6.2.5. Entropy

The Key Management Agent is assumed to have access to a suitable source of entropy. Thus operations requiring access to entropy (as well as those requiring access to key material) and performed using the Key Management API. This enables the Key Management system to manage its entropy pool, for example vary its response (for example the amount of entropy used to generate a particular ‘random’ item) depending on the current state of the entropy pool (as a function of the security policy). If this is beyond the scope of the local node (for example IoT devices), then the entropy management can be proxied to the central Key Manager.

6.3. Use cases

The following use cases expand the descriptions previously given in D4.2 to show the interactions with the Key Management Agents and the central Key Manager.

6.3.1. Generation and distribution of keys

It is the Central Manager that decides when new key material, for example an RSA key pair, is needed. It may do this either because it has been asked for a new key by a Management Agent, or it may wish to pre-generate a section of keys in advance to hide the latency of creating them. In a multi-tenant environment, there may be a choice of key managers to access, and the choice of which one will depend on the intended use of this key material. From the manager perspective, keys are kept in a special portion of its own RIB. It creates objects called 'key containers' to hold the relevant information. Note, however, that private key material is not stored in the RIB, but held in the Key Manager; the key container holds a unique ID that can be used to reference the key material held in the key manager. In the case of a public/private key pair, the public portion is stored in the RIB for easy access.

The Central Manager creates the key container for new key. Consequently, the Central Manager RIB becomes populated with associated material, in particular:

- key state
- public key matter
- the ID of the private key held by the Key Manager

We are assuming that the relevant Management Agents have access to the appropriate portions of the Manager RIB.

The process is illustrated in Figure 22 below, which also shows how Management Agents subscribe to the key container and how the local Key Management Agent obtains a copy of the private key material from the appropriate Key Manager 'KZ', selected by the RINA Manager. Note that Key Managers sit on the edge of the RINA environment, and may also be part of a larger security/privacy infrastructure.

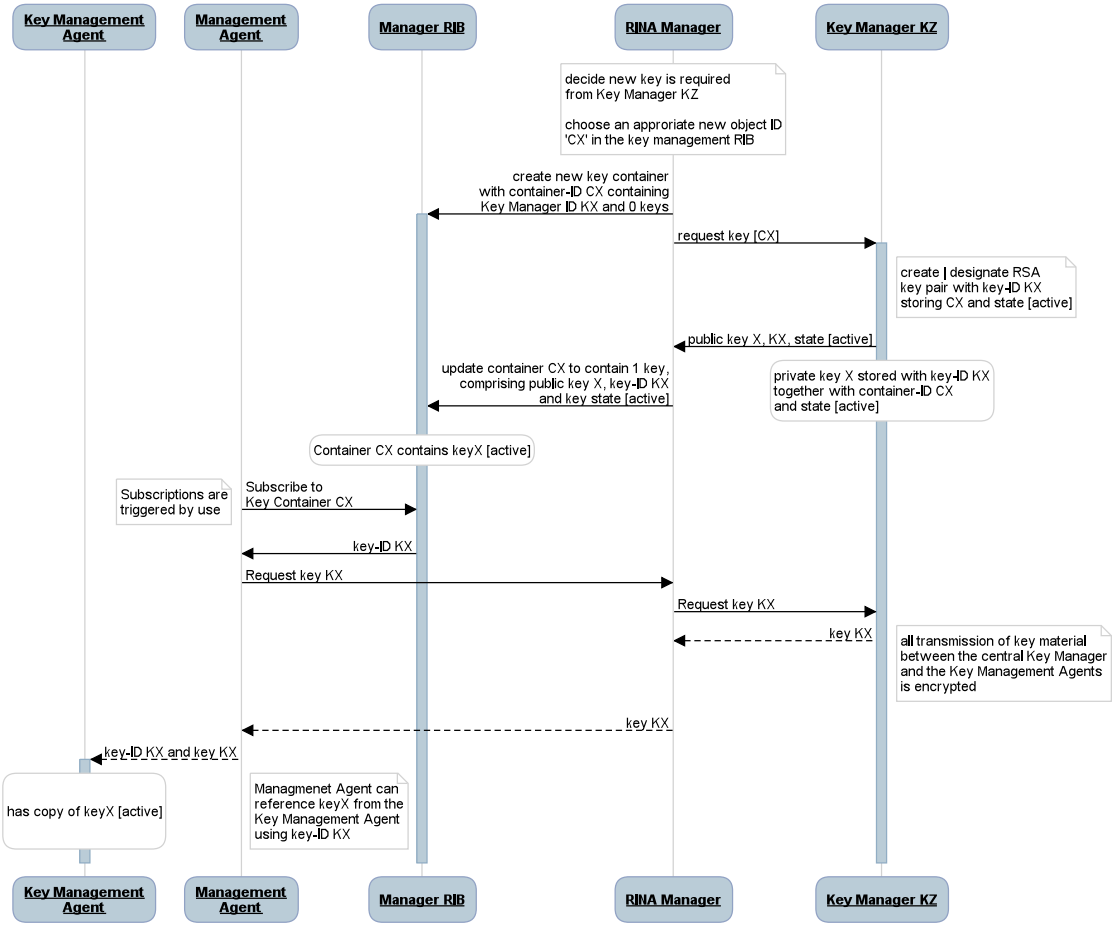

Figure 22. Creation and storage of a new public/private key pair

Note that Key Containers can contain 0, 1 or 2 keys, which is important for the next use case.

6.3.2. Periodic rotation of keys

In order to limit the computation that an attacker can deploy against a key, it is good practice to replace keys with new ones on a regular basis - this is called 'key rotation'. As this is a normal operational procedure, it should proceed without causing disturbance to other processes, such as IPCP authentication. This is made complex by the fact that many processes may require access to the same key, and many Key Management Agents may have a copy of it; we rely on the RIB notification function to inform affected nodes that they need to update their copies of the key material.

To enable this function, a key container in the RIB must be able to accommodate two keys that can be used interchangeably. To perform key rotation without disturbing normal operation, we allow a state in which both the old and the new (replacement) key are valid, then move to one

in which the old key can no longer be used to initiate new actions, but can still be used to respond, so that an action that was initiated before this state change can be completed. Finally the old key is removed from use and operation moves seamlessly to using the new one. By allowing sufficient time between state changes for the new state to propagate to all affected nodes we avoid race conditions. This is shown in Figure 23 below, which repeats the subscription step - this is shown only for one Management Agent, but can be repeated by as many as need to use this particular key; all such Management Agents will be notified of changes to the key container and will participate in the process as shown. The proxying of communications between the Key Management Agents and the Central Key Manager via the Management Agents and Central Manager is not shown for clarity, but this can be inferred by comparison with Figure 22 above.

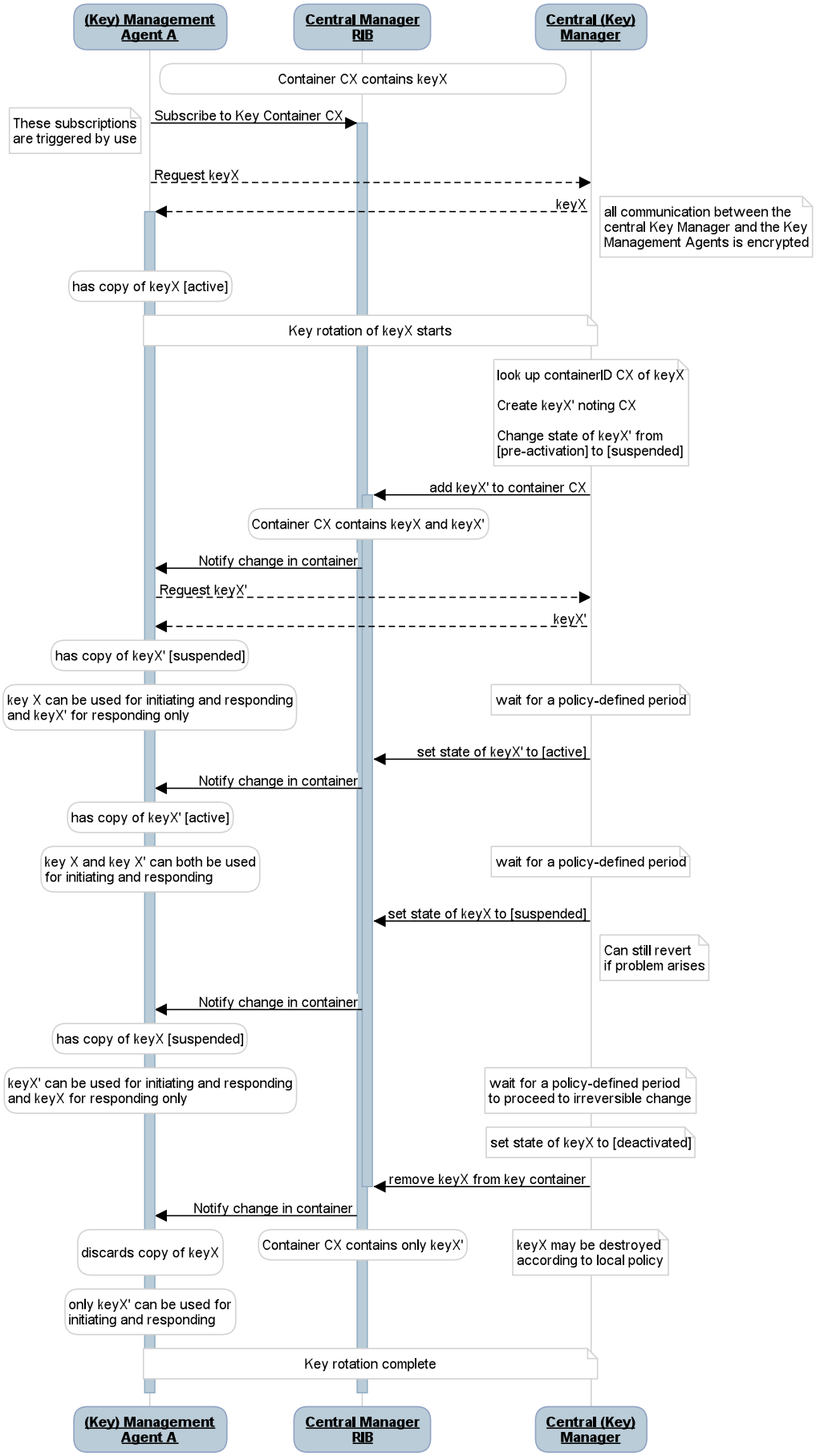

Figure 23. Rotation of a key

Note that keys are permitted to transition from the suspended to the active state. Thus roll-backs are possible should something go wrong with the procedure.

6.3.3. Password authentication

This is a simple authentication protocol, based on demonstrating access to a shared secret (the 'password'). This shared secret is key material that is held in the Key Manager and referenced via a Key Container. In order to identify this material, there needs to be a policy-defined mapping between the various M_CONNECT parameters, that is known to all the processes within the security domain. The process is shown in Figure 24.

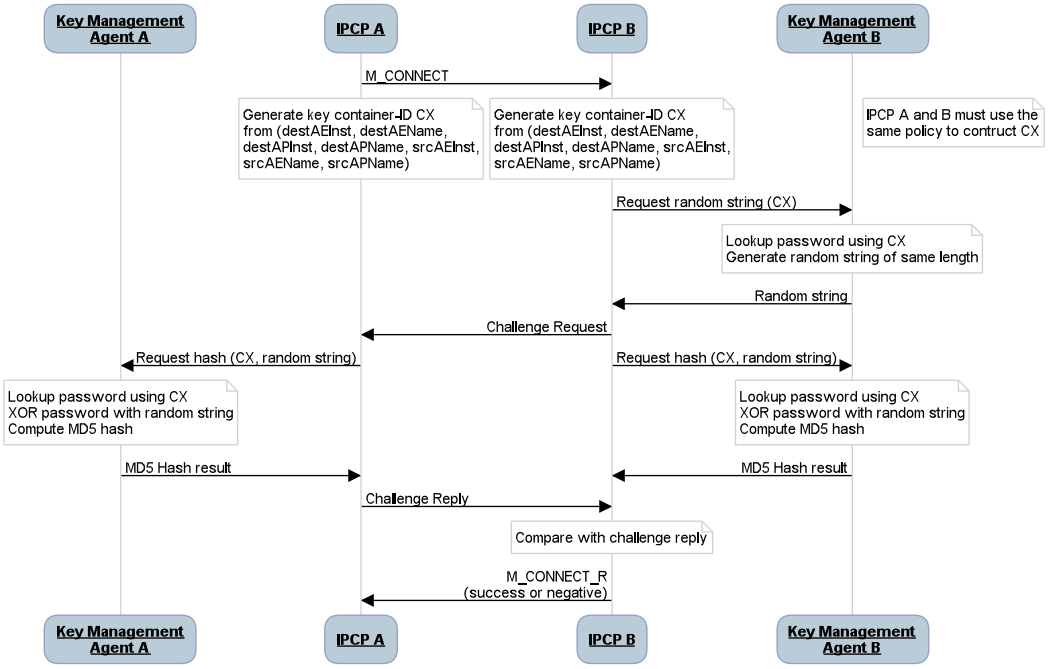

Figure 24. Password Authentication using Key Management Agent

The generation of the random string is performed by the Key Management Agent because it both requires a source of entropy and access to the shared secret (to know its length).

In the case that the local Key Management agent does not (yet) have a copy of the password, but is permitted to do so, it will request a copy from the the central key manager, as shown in Figure 25.

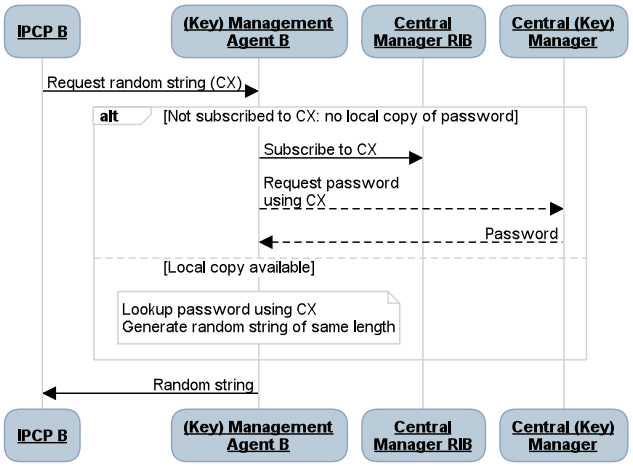

Figure 25. Password Authentication using Key Management Agent Cache

In the case that the local Key Management Agent is not permitted to have key material (due to being resident on an untrusted node), the operation is proxied to the central Key Manager, as shown in Figure 26.

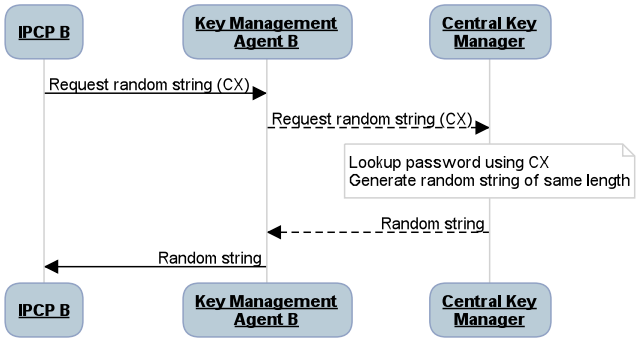

Figure 26. Password Authentication using Key Management Proxy

In both cases the API from the Key Management Agent via the Management Agent to the IPCP is the same; the implementation behind the API is determined by specific circumstances and the overall security policy.

Further applications of the Key Management function to specific use cases follow the same pattern.

6.3.4. AuthNAsymmeticKey Policy

This is quite complex, involving both a Diffie-Hellman key exchange and an RSA bi-directional authentication, so we break it here into two parts for clarity, but these should be read as one sequence. Figure 27 shows the first part, including the Diffie-Hellman key exchange, but also the interaction

with the RIB to obtain the appropriate public and private parts of the RSA keys - doing these at the same time reduces the time to complete the overall process.

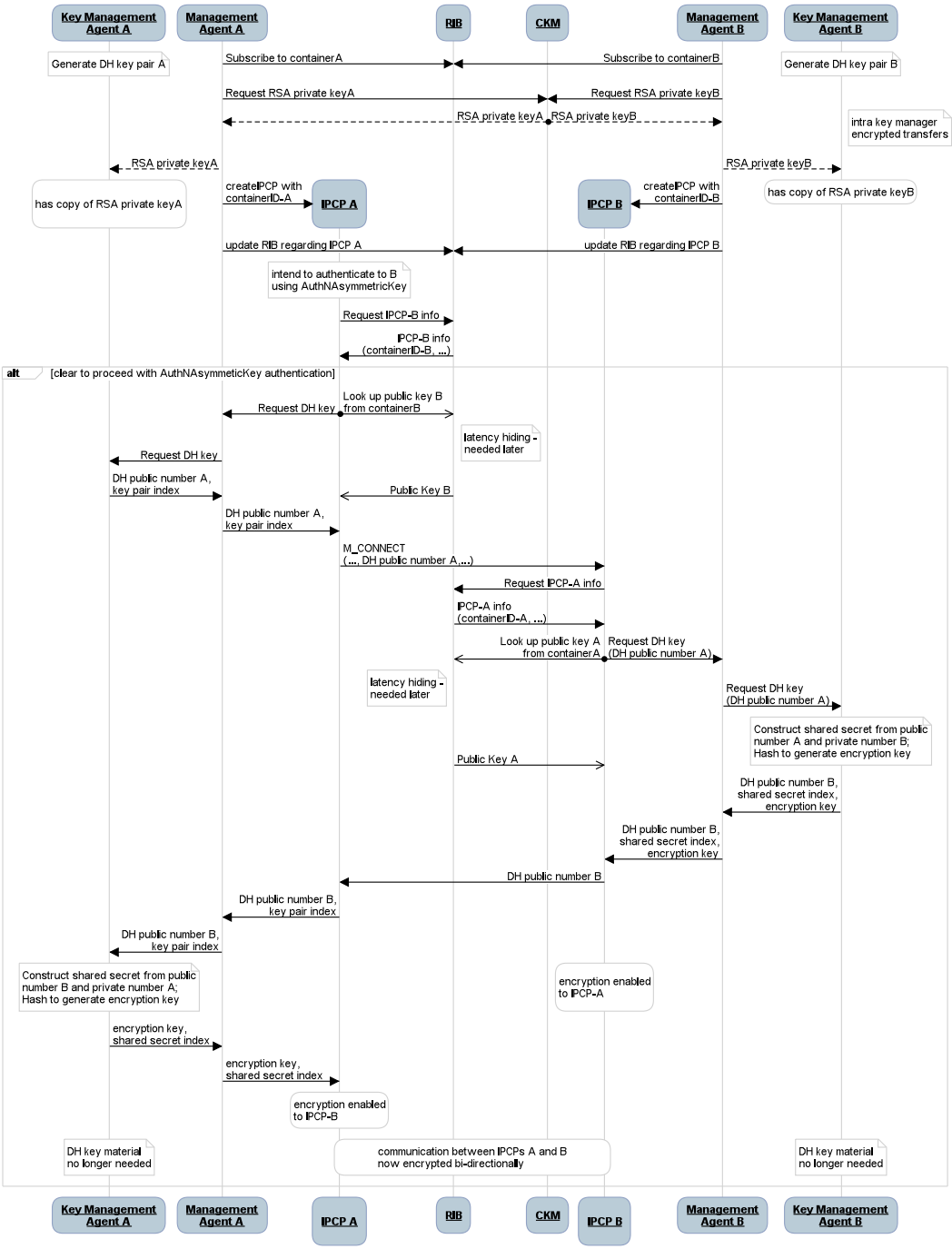

Figure 27. Password Authentication using Asymmetric Key Part 1

Figure 28 shows the second part of the sequence, which does not require any further interaction with the RIB.

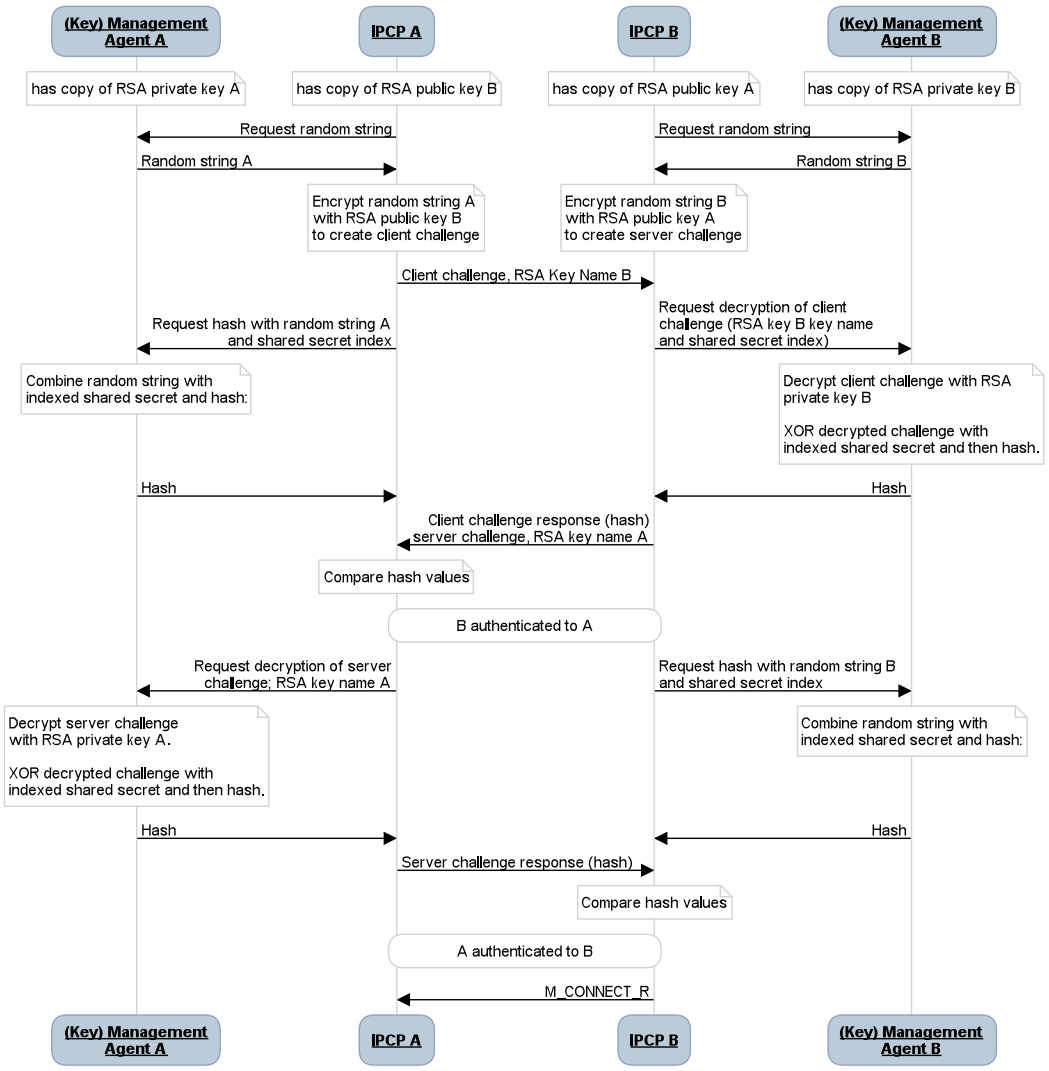

Figure 28. Password Authentication using Asymmetric Key Part 2

Note that additional information regarding the association needs to be supplied to Key Management Agent B in order to provide an audit trail.

Note that the assumption that there is a shared name space for RSA public keys does not require both Key Management Agents to belong to the same management domain; thus it is possible to establish authenticated connections between domains.

Interaction with key rotation

In Section 6.3.2 we stated that key rotation should not disturb authentication. However, there is a phase of this process in which there are two active keys available to IPCP A. In this case, one of them should be chosen at random. At the other side, the test should be performed also using a randomly chosen one of the active keys; if this fails the test should be

repeated with the other key (note this will happen 50% of the time). Where the receiver has one active key and one in another state, the active one should be tried first. Thus, key rotation may cause an extra computational burden, but does not affect the sequence of the authentication process.

Note that in the case that one of the Key Management Agents loses communication with the central Key Manager during the key rotation process, the random choice will cause authentication to always fail 50% of the time, which is easily detected.

7. Formal verification of security controls

This chapter describes formal analysis experiments that deal with threats related to communication between IPC processes in RINA architecture. The experiments consist of a formal analysis of identified security risks followed by the demonstration of found flaws using simulation model. Among many possible threats, we only consider attacks related to IPCP communication. These attacks can be directly expressed in Dolev-Yao model [Dolev1983] of the attacker. All these attacks assume that the attacker has access to a communication channel that carries traffic between communicating parties. In the case of RINA, this channel can be found at various places. It can be physical communication medium in a ShimDIF, or it can be malicious IPCP that offers its services to DIF above. The RINA security is based on the principle of isolation of layers. The minimal trust between layers is assumed so when two IPC processes at the same DIF want to communicate securely they need to protect the communication between themselves. In RINA, to protect the communication, the communicating IPC processes choose appropriate SDU protection policy that provides confidentiality and integrity. The complexity of creating a testbed environment for verification of defined security controls in protecting RINA assets led us to consider alternative means of verification. Formal methods are mathematically-based techniques for the specification, development, and verification of software aspects of digital systems. We have considered formal verification technique and applied ProVerif tool for formal analysis of RINA security and RINASim [Vesely2015] for a demonstration of identified security threats. Applying a formal tool for verification of security mechanism enables us to determine the attack traces and verify the properties of security measures applied to mitigate the security threats associated with these attacks. The results presented mainly comprise of formally verified network architecture (RINA) with respect to security. This work can be viewed as a complementary analysis to that done by Boddapati et al. [Boddapati2012] and Grasa et al. [secicc2016].

7.1. Definition of Threat Model

Before we provide a model for RINA, we demonstrate the approach and identify attacks and security controls in a simplified environment consisting only of two communication nodes connected through the one unidirectional channel. Using this simplified model, we show the principle

on how the attacks can be represented and analysed. Also, this model is used to explain basic principles of ProVerif modelling and verification. To define the threat model, we adopt the Dole-Yao definition of an attacker [Dolev1983]. The attacker is capable of a rich set of operations and active interference, namely:

- listen to any message on the network and within compromised IPCP,
- inject or modify any message on the network and within compromised IPCP, and
- delete any message in the network and from the compromised IPCP.

Depending on the threat scenario analysed, the attacker has access to various communication channels or information.

SDU protection is a component that lies at the very bottom of every IPCP. This component is responsible for protecting SDUs that are passed to underlying IPCP and to check and validate the incoming SDUs. We will analyse two SDU protection policies that are currently available in RINA implementation.

- SDUP_NONE is a default SDU Protection Policy that applies no efficient mechanism to (cryptographically) protect SDUs. This policy performs only basic SDU protection that consists of computing checksum and TTL.
- SDUP_CRYPTO is an SDU Protection Policy based on the utilisation of standard cryptographic operations, such as encryption to protect against eavesdropping and cryptographic hashing to protect integrity and authenticity of messages.

The automatic formal verification tool ProVerif is designed to verify security protocols. The language of ProVerif is based on pi-calculus. The properties are specified as assertions and correspondence assertions. Correspondence assertion has a form of $H \Rightarrow G$ and states that if some event H has been executed, then other event G have been executed before. Verification of security properties is done by translating the protocol model to Horn clauses and applying resolution procedure showing that assertions represented as queries are not derivable. When the proof fails, ProVerif provides a derivation of a fact from the clauses. ProVerif modelling language consists of a few constructs as explained in the following table.

Table 7. ProVerif Language Summary

in(c,m);P	sends the message m on a channel named c and continues as the process P
out(c,x);P	receives a message m on a channel named c, matches m with the pattern x, and continues as P
new m;P	creates a fresh name m and continues as the process P
event e(M1,...,Mn);P	executes the event e(M1, ...,Mn) and continues as the process P
if M=N then P else Q	executes P if M evaluates to the same term as M′; otherwise it executes Q.
lex x = M in P	evaluates M, matches it with the pattern X and, when the matching succeeds, continues as P
P || Q	runs the processes P and Q in parallel.
! P	runs an unbounded number of copies of the process P in parallel.

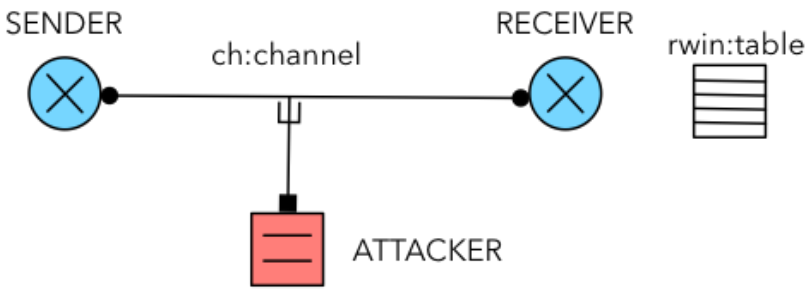

Figure 29. A Model of Two Hosts over a Direct Channel

The model and the corresponding formal specification of the simplified environment is shown in Figure 29 and Figure 30 respectively. Lines 1-5 define different types of objects used in the model. Type msg models messages, type sdu represents SDU objects that pack messages before they can be transmitted on the channel, type key represents encryption keys used in SDU protection. Type id represents a collection of identifiers used to number messages, and the receiver uses type tag for recording delivered messages. Lines 6-8 define events that are used in queries during the automated analysis. Name ch is used for identification of a communication channel that links sender to receiver. Table rwin models replay window. It stores all received messages and enables to test if the receiver gets the duplicated message by searching rwin table for its id. There are four queries which will be discussed later. SDU protection operations are represented by constructor sdu_protect and the corresponding destructor sdu_check. To analyse this scenario, we define two keys. KEY is a private key while NULL key is a public key and it also represents the situation when no key

is used, or the key is known to the attacker, e.g., because all nodes use the single key within a DIF, and the attacker is a member of this DIF.

```
001 type msg.
002 type sdu.
003 type key.
004 type id.
005 type tag.
006 event evt_send(msg).
007 event evt_accept(msg).
008 event evt_discard(msg).
009
010 free ch:channel.
011 table rwin(id, tag).
012
013 (* C1: Message cannot be inserted. *)
014 query m:msg; event(evt_accept(m)) ⟹ event(evt_send(m)).
015 (* C2: Message cannot be modified *)
016 fun modify_msg(msg) : msg [data].
017 query event(evt_accept(modify_msg(MSG))).
018 (* C3: Message cannot be read *)
019 query attacker(MSG).
020 (* C4: Message cannot be replayed *)
021 query inj-event(evt_accept(MSG)) ⟹ inj-event(evt_send(MSG)).
022
023 (* Abstract representation of sdu protection function. *)
024 fun sdu_protect(key, id, msg):sdu.
025 reduc forall k:key, i:id, m:msg; sdu_check(k, sdu_protect(k, i, m)) = (i, m).
026
027 free KEY: key [private].
028 free NULL : key.
029
030 free MSG : msg [private].
031
032 let sender(k:key) =
033     new i:id;
034     let s = sdu_protect(k, i, MSG) in
035     event evt_send(MSG);
036     out(ch, s).
037
038 let receiver(k:key) =
039     in(ch, s:sdu);
040     let (i:id, m:msg) = sdu_check(k, s) in
041     new t:tag;  insert rwin(i, t);
042     get rwin(= i, t') suchthat t ≠ t'
043     in    event evt_discard(m)
044     else  event evt_accept(m).
045
```

Figure 30. Formal Specification of Simplified Scenario

To verify the model, message MSG represents a private message that is used to model the communication between parties. Communicating parties are modelled as two processes using the let-statement. Sender process (lines 32-36) takes the message MSG applies for the protection according to the key specified as a parameter k and sends the protected message to the channel ch. Also, the event evt_send is recorded on line 35. Receiver process (lines 38-44) reads the packet from the channel ch and

unpack the message if the security check is done. Lines 41-42 represent replay window operations. If replay window already contains the id of the received message, then the message is discarded (line 43). Otherwise, the message is accepted for delivery (line 44).

To analyse the model when no protection is applied the process is defined as follows:

```
process
    (!sender(NULL))|(!receiver(NULL))
```

This scenario means that the sender and receiver use known or none key. First, we analyse this model in ProVerif tool that provides us with the following output:

```
#ProVerif sdu_channel.pv | grep RESULT

RESULT inj-event(evt_accept(MSG[])) ==> inj-event(evt_send(MSG[])) cannot
 be proved.
RESULT (but event(evt_accept(MSG[])) ==> event(evt_send(MSG[])) is true.)
RESULT not attacker(MSG[]) is false.
RESULT not event(evt_accept(modify_msg(MSG[]))) is false.
RESULT event(evt_accept(m_952)) ==> event(evt_send(m_952)) is false.
```

Results are provided for all queries. Note that the results are presented in the reverse order than the order in which queries appear in the specification. The last result corresponds to query annotated as a C1 attack. Last three results are false, which means that the specified properties do not hold, and the tool was able to find counterexamples. We can further analyse these counterexamples to see the problems found. A counter example may correspond to attack. We begin with the simplest query, which represents attack C3 that was evaluated as:

```
RESULT not attacker(MSG[]) is false.
```

The query captures the fact that malicious IPCP eavesdrops communication. This query is evaluated by the tool trying to find the situation when the attacker can obtain the content of the message MSG. If such situation cannot be considered in the model, then the result is true.

Otherwise, the result is false and other information explaining how the attacker can get MSG is provided, which is our case:

```
-- Query not attacker(MSG[])
Completing...
Starting query not attacker(MSG[])
goal reachable: attacker(MSG[])
Abbreviations:
i_722 = i_32[!1 = @sid_716]
1. The message sdu_protect(NULL[],i_722,MSG[]) may be sent to the attacker
 at output {5}.
attacker(sdu_protect(NULL[],i_722,MSG[])).
2. The attacker initially knows NULL[].
attacker(NULL[]).
3. By 2, the attacker may know NULL[].
By 1, the attacker may know sdu_protect(NULL[],i_722,MSG[]).
Using the function sdu_check the attacker may obtain (i_722,MSG[]).
attacker((i_722,MSG[])).
4. By 3, the attacker may know (i_722,MSG[]).
attacker(MSG[]).

new i_32 creating i_724 at {2} in copy a_723
event(evt_send(MSG)) at {4} in copy a_723
out(ch, sdu_protect(NULL,i_724,MSG)) at {5} in copy a_723

The attacker has the message MSG.
A trace has been found.
RESULT not attacker(MSG[]) is false.
```

The information contains derivation and trace sections that correspond to the situation when the attacker can obtain a plain message MSG. Derivation consists of four steps that need to be done to see the message by the attacker. Step 1 stands for sending the protected message with NULL key to the channel. Step 2 expresses that the attacker knows NULL key. Step 3 expresses that when the attacker knows the key she may obtain the message from the protected SDU. Step 4 is an application of the data access operation to get the message from the tuple (message, id). Trace block is a sequence of significant events in the system to accomplish the attack. It is easy to see that the problem is in the usage of the known key in SDU protection. This way also the use case when no protection is applied can be analysed.

Next, we analyse the second query that corresponds to attack C2:

```
RESULT not event(evt_accept(modify_msg(MSG[]))) is false.
```

The query expresses that it is not possible for an attacker to modify the message so that the receiver accepts the message. It is encoded as testing if the event evt_accept for modified MSG as a parameter can be reached. The result is false that means we have a counterexample showing how the attacker can act to perform this attack successfully:

```
-- Query not event(evt_accept(modify_msg(MSG[])))
Completing..
Starting query not event(evt_accept(modify_msg(MSG[])))
goal reachable: end(evt_accept(modify_msg(MSG[])))

1. The message sdu_protect(NULL[],i_925,MSG[]) may be sent to the attacker
 at output {5}.
attacker(sdu_protect(NULL[],i_925,MSG[])).
2. The attacker initially knows NULL[].
attacker(NULL[]).
3. By 2, the attacker may know NULL[].
By 1, the attacker may know sdu_protect(NULL[],i_925,MSG[]).
Using the function sdu_check the attacker may obtain (i_925,MSG[]).
attacker((i_925,MSG[])).
4. By 3, the attacker may know (i_925,MSG[]).
Using the function 2-proj-2-tuple the attacker may obtain MSG[].
attacker(MSG[]).
5. By 4, the attacker may know MSG[].
Using the function modify_msg the attacker may obtain modify_msg(MSG[]).
attacker(modify_msg(MSG[])).
6. The attacker has some term i_922.
attacker(i_922).
7. By 2, the attacker may know NULL[].
By 6, the attacker may know i_922.
By 5, the attacker may know modify_msg(MSG[]).
Using the function sdu_protect the attacker may obtain
 sdu_protect(NULL[],i_922,modify_msg(MSG[])).
attacker(sdu_protect(NULL[],i_922,modify_msg(MSG[]))).
8. The message sdu_protect(NULL[],i_922,modify_msg(MSG[])) that the
 attacker may have by 7 may be received at input {7}.
So event evt_accept(modify_msg(MSG[])) may be executed at {12}.
end(evt_accept(modify_msg(MSG[]))).

new i_32 creating i_929 at {2} in copy a_927
event(evt_send(MSG)) at {4} in copy a_927
out(ch, sdu_protect(NULL,i_929,MSG)) at {5} in copy a_927
```

```
in(ch, sdu_protect(NULL,a_926,modify_msg(MSG))) at {7} in copy a_928
new t creating t_947 at {9} in copy a_928
insert rwin(a_926,t_947) at {10} in copy a_928
get: else branch taken at {13} in copy a_928
event(evt_accept(modify_msg(MSG))) at {12} in copy a_928
The event evt_accept(modify_msg(MSG)) is executed.
A trace has been found.
RESULT not event(evt_accept(modify_msg(MSG[]))) is false.
```

The attacker can intercept message MSG and then modify it so that the receiver accepts this modified message. It is because the attacker knows all necessary operations and NULL is used to protect the communication.

Next attack that we will analyse is the possibility to insert a new message to the communication channel that is accepted by the receiver. This corresponds to attack C1.

```
-- Query event(evt_accept(m_952)) ==> event(evt_send(m_952))
Completing...
Starting query event(evt_accept(m_952)) ==> event(evt_send(m_952))
goal reachable: attacker(m_1142) -> end(evt_accept(m_1142))
1. We assume as hypothesis that
attacker(m_1150).
2. The attacker has some term i_1147.
attacker(i_1147).
3. The attacker initially knows NULL[].
attacker(NULL[]).
4. By 3, the attacker may know NULL[].By 2, the attacker may know i_1147.
 By 1, the attacker may know m_1150. Using the function sdu_protect the
 attacker may obtain sdu_protect(NULL[],i_1147,m_1150).
attacker(sdu_protect(NULL[],i_1147,m_1150)).
5. The message sdu_protect(NULL[],i_1147,m_1150) that the attacker may
 have by 4 may be received at input {7}. So event evt_accept(m_1150) may
 be executed at {12}.
end(evt_accept(m_1150)).

in(ch, sdu_protect(NULL,a_1152,a_1151)) at {7} in copy a_1153
new t creating t_1170 at {9} in copy a_1153
insert rwin(a_1152,t_1170) at {10} in copy a_1153
get: else branch taken at {13} in copy a_1153
event(evt_accept(a_1151)) at {12} in copy a_1153
The event evt_accept(a_1151) is executed. A trace has been found.
RESULT event(evt_accept(m_952)) ==> event(evt_send(m_952)) is false.
```

Derivation consists of 5 steps. Four steps are related to fabricating a message. Step 5 expresses sending the message to the receiver. Finally, we look at the analysis of attack C4. This attack assumes that receiver accepts replicated/replayed messages. The specification of the property related to this attack is expressed as the injective correspondence between sending and accepting events. This correspondence says that the message has to be accepted if the real sender sent it.

```
query inj-event(evt_accept(MSG)) ==> inj-event(evt_seng(MSG))
```

This query asserts that, for every occurrence of the accept, there is a distinct send event occurred earlier. Note that the result of this query was that it cannot be proved. If we look at the associated information, we may find the answer. The derivation and trace are as follows:

```
-- Query inj-event(evt_accept(MSG[])) ==> inj-event(evt_send(MSG[]))
Completing...
Starting query inj-event(evt_accept(MSG[])) ==> inj-event(evt_send(MSG[]))
goal reachable: begin(evt_send(MSG[]), @sid = @sid_249, @occ4 = @occ_cst)
 -> end(endsid_250,evt_accept(MSG[]))
Abbreviations: i_270 = i_32[!1 = @sid_257]

1. The event evt_send(MSG[]) (with environment @sid = @sid_257,
 @occ4 = @occ_cst) may be executed at {4}. So the message
 sdu_protect(NULL[],i_270,MSG[]) may be sent to the attacker at output
 {5}.
attacker(sdu_protect(NULL[],i_270,MSG[])).
2. The attacker initially knows NULL[].
attacker(NULL[]).
3. By 2, the attacker may know NULL[].By 1, the attacker may know
 sdu_protect(NULL[],i_270,MSG[]).Using the function sdu_check the attacker
 may obtain (i_270,MSG[]).
attacker((i_270,MSG[])).
4. By 3, the attacker may know (i_270,MSG[]).Using the function 2-proj-2-
tuple the attacker may obtain MSG[].
attacker(MSG[]).
5. The attacker has some term i_266.
attacker(i_266).
6. By 2, the attacker may know NULL[].By 5, the attacker may know i_266.
 By 4, the attacker may know MSG[]. Using the function sdu_protect the
 attacker may obtain sdu_protect(NULL[],i_266,MSG[]).
attacker(sdu_protect(NULL[],i_266,MSG[])).
```

```
7. The message sdu_protect(NULL[],i_266,MSG[]) that the attacker may
 have by 6 may be received at input {7}.So event evt_accept(MSG[]) may be
 executed at {12} in session endsid_269.
end(endsid_269,evt_accept(MSG[])).

new i_32 creating i_273 at {2} in copy a_272
event(evt_send(MSG)) at {4} in copy a_272
out(ch, sdu_protect(NULL,i_273,MSG)) at {5} in copy a_272
in(ch, sdu_protect(NULL,a_271,MSG)) at {7} in copy a
new t creating t_290 at {9} in copy a
insert rwin(a_271,t_290) at {10} in copy a
get: else branch taken at {13} in copy a
event(evt_accept(MSG)) at {12} in copy a

The event evt_accept(MSG) is executed in session a.
A trace has been found.
I am now trying to reconstruct a trace that falsifies injectivity.
Could not find a trace that contradicts injectivity.

RESULT inj-event(evt_accept(MSG[])) ==> inj-event(evt_send(MSG[])) cannot
 be proved.
RESULT (but event(evt_accept(MSG[])) ==> event(evt_send(MSG[])) is true.)
```

In the derivation, steps 1-4 represents how the attacker gets knowledge about MSG. In steps 5 and 6, the attacker creates a correctly protected SDU with message MSG and a unique id. This SDU is sent to the receiver in step 7. Because the id is different from the MSG is accepted by the receiver. Steps 5-7 can be performed multiple times thus it is possible to replay the SDU containing the message MSG.

The tool has difficulties in proving this property in the presented form. However, it suggests that non-injective correspondence is satisfied. Indeed, the attacker must first know MSG to replay it thus at least one evt_send event has to occur before evt_accept. It was the last property analysed for NULL key. Next, we will analyse the case when a secret key is used in SDU protection.

To analyse the model with cryptographic SDU protection both processes are created with KEY as a parameter, which is encoded as follows:

```
process (!sender(KEY))|(!receiver(KEY))
```

Here, the key is only known to sender and receiver but not the attacker. Because of that, the results of the verification are all true except the injective correspondence:

```
RESULT inj-event(evt_accept(MSG[])) ==> inj-event(evt_send(MSG[])) cannot
 be proved.
RESULT (but event(evt_accept(MSG[])) ==> event(evt_send(MSG[])) is true.)
RESULT not attacker(MSG[]) is true.
RESULT not event(evt_accept(modify_msg(MSG[]))) is true.
RESULT event(evt_accept(m_858)) ==> event(evt_send(m_858)) is true.
```

By interpretation of results we get that the attacker cannot:

- Insert a new message that is accepted by the receiver (threat C1)
- Modify an existing message that is then accepted by the receiver (threat C2)
- Eavesdrop any message send by the sender (threat C3)

The impossibility of replay a message from the sender that is accepted by the receiver (threat C4) was again not proved by the ProVerif, and thus we need to analyse the provided information manually:

```
-- Query inj-event(evt_accept(MSG[])) ==> inj-event(evt_send(MSG[]))
Completing...
Starting query inj-event(evt_accept(MSG[])) ==> inj-event(evt_send(MSG[]))
goal reachable: begin(evt_send(MSG[]), @sid = @sid_234, @occ4 = @occ_cst)
 -> end(endsid_235,evt_accept(MSG[]))
Abbreviations:i_245 = i_32[!1 = @sid_240]
1. The event evt_send(MSG[]) (with environment @sid = @sid_240,
 @occ4 = @occ_cst) may be executed at {4}.So the message
 sdu_protect(KEY[],i_245,MSG[]) may be sent to the attacker at output {5}.
attacker(sdu_protect(KEY[],i_245,MSG[])).
2. The message sdu_protect(KEY[],i_245,MSG[]) that the attacker may have
 by 1 may be received at input {7}.So event evt_accept(MSG[]) may be
 executed at {12} in session endsid_244.
end(endsid_244,evt_accept(MSG[])).

new i_32 creating i_247 at {2} in copy a_246
event(evt_send(MSG)) at {4} in copy a_246
out(ch, sdu_protect(KEY,i_247,MSG)) at {5} in copy a_246
in(ch, sdu_protect(KEY,i_247,MSG)) at {7} in copy a
new t creating t_264 at {9} in copy a
```

```
insert rwin(i_247,t_264) at {10} in copy a
get: else branch taken at {13} in copy a
event(evt_accept(MSG)) at {12} in copy a

The event evt_accept(MSG) is executed in session a.
A trace has been found. I am now trying to reconstruct a trace that
 falsifies injectivity.
Could not find a trace that contradicts injectivity.
RESULT inj-event(evt_accept(MSG[])) ==> inj-event(evt_send(MSG[])) cannot
 be proved.
```

The listing shows that ProVerif was able to assert reachability of the injective correspondence but could not prove it. Relevant information is that the tool was not able to find the trace that contradicts injectivity. correspondence. Note, that finding a trace means to identify the sequence of actions that lead to the attack. Here, the attack was not identified, but because ProVerif reasoning system is incomplete the proof could not be found, and we need to check this case manually, or we need to rewrite (simplify) the specification to help the tool in the search for the proof.

All of the analysed attacks were mitigated by the SDU protection security control that protects confidentiality and integrity of SDU. The replay window mechanism is applied to mitigate threat C4. In the next section, we extend our model to incorporate the abstraction of basic RINA communication mechanism. Using this RINA model we show how the identified attacks can be realised in this model without SDU protection applied. After that, we show proofs of security and integrity of SDU protection policy.

7.2. RINA Network Model

In this section, we will build a more complex model that integrates an abstraction of RINA process behaviour related to the attacks being analysing. The aim is to show how the attacks described in the previous section can be carried out in RINA and how these attacks can be avoided by applying the cryptographic SDU Protection policy.

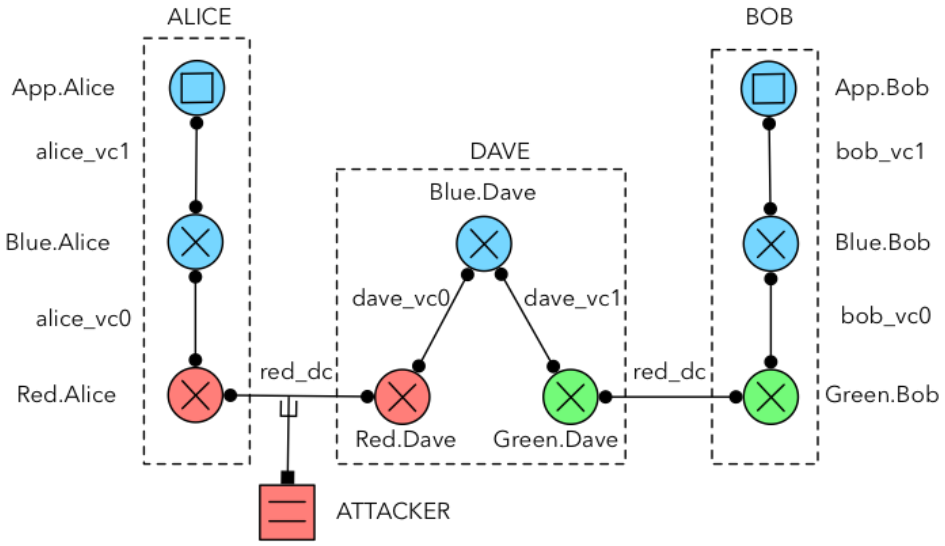

Figure 31. A Network Topology of Analysed Scenario

Potential attacks on the network that originate from the N-1 level DIF are analysed using the network topology as shown in Figure 3. There are three nodes. ALICE and BOB are considered regular hosts that run RINA applications. DAVE is a router node that provides connectivity for Blue DIF by interconnecting supporting Red and Green DIFs. An attacker can access Red DIF. There are three DIFs in this scenario:

- Blue – this DIF represents a common layer for all nodes in this scenario. Applications directly use this DIF for communication.
- Red – this DIF is a local DIF that interconnects ALICE and DAVE. It is also supporting DIF for Blue DIF. Red DIF uses broadcast communication medium.
- Green – this DIF is another local DIF that interconnects DAVE and BOB. It is also supporting DIF for Blue DIF.

A formal model of RINA and SDU protection are encoded in cypto.pvl and rina.pvl (see Section C.1), respectively. Library crypto.pvl contains cryptographic operations that are utilised to represent security mechanisms implemented in SDU protection module of RINA. Library rina.pvl accounts for a simplified specification of RINA IPCP operations, such as, send and receive, flow allocation, enrollment, authentication and application communication.

Library rina.pvl defines types and operations of our RINA model. The source code is shown in Figure 4. Lines 1-9 defines types for the objects of the model. They are followed by three groups of events. These events

correspond to reception, acceptance and drop of the SDU (lines 11-13), PDU (lines 15-17) and CDAP (lines 19-21) messages. Constructors to make PDU message are presented on lines 29 and 37. These constructors enable to create a management PDU and data PDU provided the correct parameters. A list of destructors for data PDU is given on lines 38-40. These are used to access the content of the PDU. Similarly, SDU constructors are defined (lines 44 and 48). MakeDifSdu is employed in IPCP to create SDU that encapsulates PDUs. MakeDafSdu is used by the application process to encapsulate CDAP message in SDU. Line 57 contains a constructor that defines IPC process, which consists of DIF name, process address and north-bound and south-bound channels. Lines 65 contains a constructor that defines Shim IPC process, which consists of a DIF name, process address, north-bound channel and media channel. Lines 67-80 provides three types of destructor to provide access to individual fields of IPC structures. The rest of the library script contains specification of IPC behaviour:

- SendCdap function (lines 95-99) implements the functionality of DAF IPC process that takes CDAP message, applies necessary operations on it and sends it to the underlying IPC process.
- ReceiveCdap function (lines 105-110) reads the SDU from the underlying IPC process applies SDU policy checks and forwards the CDAP message to the provided channel. Also, CdapReceiveEvent is raised when the processing of the message is over. If SDU protection checking fails, the SDU is dropped, and the SduDropEvent is raised.
- SendSdu function encapsulates provided SDU to PDU and sends it to the specified target address using underlying IPCP process. It performs SDU protection before the data are passed down to the supporting IPC process. Event PduSendEvent is raised before the data are handed over the underlying IPC process.
- ReceiveSdu reads SDU from the specified channel and applies SDU protection verification procedure. If the SDU is accepted then the PDU is unencapsulated from it, and the target address is checked. If it matches the local address, the PDU is processed, and PduAcceptEvent is raised. Otherwise, the SduDropEvent or PduDropEvent is raised.

These four functions encode the behaviour of IPC processes that can be used to build the formal model of a RINA network suitable for high-level analysis of security properties. In the formal model, the communication

channels represent the connection between IPCPs in the same DIF as well as connections between IPCPs in different DIFs within a host boundary. For instance, red_dc is a channel representing communication medium of RED DIF. Channel alice_vc1 connects Alice client application to Alice's IPCP in Blue DIF.

Script of a formal model of analyzed network (some lines were omitted)

```
001 free red dc : channel.
002 free green dc: channel [private].

005 const RED DIF : Dif .
006 const GREEN DIF : Dif .
007 const BLUE DIF : Dif .

009 const ALICE : Host.
010 const ALICE IPC BLUE : Ipcp.
011 const ALICE IPC RED : Ipcp.
012 const ALICE BLUE : Address.
013 const ALICE RED : Address.
014 free alice vc1:channel [private].
015 free alice vc0:channel [private].

017 const BOB : Host.
018 const BOB IPC BLUE : Ipcp.
019 const BOB IPC GREEN : Ipcp.
020 const BOB BLUE : Address.
021 const BOB GREEN : Address.
022 free bob vc1:channel [private].
023 free bob vc0:channel [private].

025 const DAVE : Host.
026 const DAVE IPC BLUE: Ipcp.
027 const DAVE IPC RED: Ipcp.
028 const DAVE IPC GREEN: Ipcp.
029 const DAVE BLUE : Address.
030 const DAVE RED : Address.
031 const DAVE GREEN : Address.
032 free dave vc0 : channel [private].
033 free dave vc1 : channel [private].

035 const ALICE CLIENT : Application.
036 const BOB SERVER : Application.

037 free bobServerCh : channel [private].
```

```
039 const SDUP NONE : SduProtection.
040 const SDUP CRYPTO : SduProtection [private].

043 free CDAP MESSAGE : Cdap [private].

045 fun ModifyPdu (Pdu) : Pdu [data].
046 fun ModifyCdap (Cdap) : Cdap [data].

072 process
073   let alice_blue = MakeIpcProcess (BLUE_DIF, ALICE_BLUE, alice_vc1) in
074   let alice_red = MakeShimProcess (RED_DIF, ALICE_RED, alice_vc0) in
075   let bob_blue = MakeIpcProcess(BLUE_DIF, BOB_BLUE, bob_vc1) in
076   let bob_green = MakeShimProcess(GREEN_DIF, BOB_GREEN, bob_vc0) in
077     (!SendCdap(ALICE_CLIENT, alice_blue, SDUP_CRYPTO, CDAP_MESSAGE))
078   | (!(in(alice_vc1,s:Sdu);
 SendSdu(alice_blue,s,BOB_BLUE,SDUP_CRYPTO,alice_vc0)))
079   | (!(in(alice_vc0,s:Sdu);
 SendSdu(alice_red,s,BOB_GREEN,SDUP_NONE,red_dc)))
080   | (!(ReceiveSdu(bob_green,SDUP_NONE,red_dc)))
081   | (!(ReceiveSdu(bob_blue, SDUP_CRYPTO, bob_vc0)))
082   | (!ReceiveCdap(BOB_SERVER, bob_blue, SDUP_CRYPTO, bobServerCh))
```

The formal model that corresponds to RINA network topology is listed in Script of a formal model of analyzed network (some lines were omitted). Lines 1-2 defines Shim DIF media. Channel red_dc is marked as public and thus, this channel can be observed by an attacker. Lines 5-7 contain DIF name definitions. Lines 9-15 contain a definition of Alice host. A host definition consists of specifying the identity of the host, enumeration of host's IPC processes, their addresses and interconnecting channels. Bob host is defined on lines 17-23. Router host called Dave is defined on lines 25-33. Application processes are identified as ALICE_CLIENT and BOB_SERVER on lines 35 and 36, respectively. All messages received by the server are collected in channel bobServerCh (line 37). Two SDU protection policies are given on lines 39 and 40. SDUP_NONE represents a default policy, while SDUP_CRYPTO stands for the cryptographic SDU protection policy. Message CDPA_MESSAGE (line 43) will be used in analysis and accounts for a legitimate CDAP created by the Alice client application. Functions ModifyPdu and ModifyCdap are helpers employed in the analysis of C2 and C8 attacks, respectively. Lines 72-82 creates all IPC processes and executes them by calling appropriate functions from rina.pvl. Note that only unidirectional communication is modelled as it is

sufficient to analyse the attack scenarios. If necessary, the model can be extended to provide bidirectional communication easily.

7.3. Security Analysis

Security analysis is done by showing that the formal model of RINA network satisfies the specified security properties. These security properties are related to potential attacks. For each attack, the corresponding security property claims the impossibility to carry out this attack. It should be noted that the potential threats/attacks (A1 to A8, B1-B25, and C1-C9), the risk assessment of these threats and treatment actions or security controls to put in place were fully described in deliverable D4.1. Potential attacks of 'A' type on the network that originate from an AP in the DAF where either the sending or receiving AP can be malicious. Possible attacks of 'B' type that originate within the N-level DIF or from an IPCP joining the N-level DIF. Potential attacks of 'C' type on the network that originate from the N-1-level DIF.

Possible attacks of 'C' type and corresponding security properties are as follows:

- Attack C1 - Malicious IPCP fabricates PDUs from DIF above.

```
query p:Pdu; event(PduAcceptEvent(BOB_BLUE, p)) ==>
 event(PduSendEvent(ALICE_BLUE,p)).
```

- Attack C2- Malicious IPCP modifies PDUs from DIF above.

```
query sp:SduProtection;
  let pdu =
 MakeDataPdu(BLUE_DIF,ALICE_BLUE,BOB_BLUE,MakeDafSdu(sp,CDAP_MESSAGE))
  in  event (PduAcceptEvent(BOB_BLUE, ModifyPdu(pdu))).
```

- Attack C3 - Malicious IPCP eavesdrops PDU from DIF above.

```
query sp:SduProtection;
  let pdu =
 MakeDataPdu(BLUE_DIF,ALICE_BLUE,BOB_BLUE,MakeDafSdu(sp,CDAP_MESSAGE))
  in  attacker(pdu).
```

- Attack C4 -Malicious IPCP replays PDUs from DIF above.

```
query sp:SduProtection;
  let pdu =
 MakeDataPdu(BLUE_DIF,ALICE_BLUE,BOB_BLUE,MakeDafSdu(sp,CDAP_MESSAGE))
  in inj-event (PduAcceptEvent(BOB_BLUE, pdu))
     ==> inj-event (PduSendEvent(ALICE_BLUE, pdu)).
```

- Attack C7 - Malicious IPCP fabricates CDAP message.

```
query m:Cdap; event(CdapReceiveEvent(BOB_SERVER, m))
              ==> event(CdapSendEvent(ALICE_CLIENT, m)).
```

- Attack C8 - Malicious IPCP modifies CDAP messages.

```
query event(CdapReceiveEvent(BOB_SERVER, ModifyCdap(CDAP_MESSAGE))).
```

- Attack C9 - Malicious IPCP eavesdrop CDAP messages.

```
query attacker(CDAP_MESSAGE).
```

All these security properties are analysed automatically by the ProVerif tool for two models. In the first model, adequate security controls are not applied. In the second model, the security controls represented by the Crypto SDU Protection Policy protects RINA against 'C' type attacks. The following output is from the analysis of the secured model:

```
-- Query not
 event(CdapReceiveEvent(BOB_SERVER,ModifyCdap(CDAP_MESSAGE[])))
Completing...
Starting query not
 event(CdapReceiveEvent(BOB_SERVER,ModifyCdap(CDAP_MESSAGE[])))
RESULT not event(CdapReceiveEvent(BOB_SERVER,ModifyCdap(CDAP_MESSAGE[])))
 is true.
-- Query event(CdapReceiveEvent(BOB_SERVER,m)) ==>
 event(CdapSendEvent(ALICE_CLIENT,m))
Completing...
Starting query event(CdapReceiveEvent(BOB_SERVER,m)) ==>
 event(CdapSendEvent(ALICE_CLIENT,m))
goal reachable: begin(CdapSendEvent(ALICE_CLIENT,CDAP_MESSAGE[])) ->
 end(CdapReceiveEvent(BOB_SERVER,CDAP_MESSAGE[]))
```

```
RESULT event(CdapReceiveEvent(BOB_SERVER,m)) ==>
 event(CdapSendEvent(ALICE_CLIENT,m)) is true.
-- Query not attacker_Cdap(CDAP_MESSAGE[])
Completing...
Starting query not attacker_Cdap(CDAP_MESSAGE[])
RESULT not attacker_Cdap(CDAP_MESSAGE[]) is true.
-- Query not attacker_Pdu(MakeDataPdu(BLUE_DIF,ALICE_BLUE,BOB_BLUE,
 MakeDafSdu(sp_4164,CDAP_MESSAGE[])))
Completing...
Starting query not attacker_Pdu(MakeDataPdu(BLUE_DIF,ALICE_BLUE,BOB_BLUE,
  MakeDafSdu(sp_4164,CDAP_MESSAGE[])))
RESULT not attacker_Pdu(MakeDataPdu(BLUE_DIF,ALICE_BLUE,BOB_BLUE,
  MakeDafSdu(sp_4164,CDAP_MESSAGE[]))) is true.
-- Query not event(PduAcceptEvent(BOB_BLUE,ModifyPdu(
  MakeDataPdu(BLUE_DIF,ALICE_BLUE,BOB_BLUE,
  MakeDafSdu(sp_5465,CDAP_MESSAGE[]))))))
Completing...
Starting query not event(PduAcceptEvent(BOB_BLUE,ModifyPdu(
  MakeDataPdu(BLUE_DIF,ALICE_BLUE,BOB_BLUE,
  MakeDafSdu(sp_5465,CDAP_MESSAGE[]))))))
RESULT not event(PduAcceptEvent(BOB_BLUE, ModifyPdu(
  MakeDataPdu(BLUE_DIF,ALICE_BLUE,BOB_BLUE,
  MakeDafSdu(sp_5465,CDAP_MESSAGE[]))))))) is true.
-- Query event(PduAcceptEvent(BOB_BLUE,p)) ==>
 event(PduSendEvent(ALICE_BLUE,p))
Completing...
Starting query event(PduAcceptEvent(BOB_BLUE,p)) ==>
 event(PduSendEvent(ALICE_BLUE,p))
goal reachable: begin(PduSendEvent(ALICE_RED,
    MakeDataPdu(RED_DIF,ALICE_RED,BOB_GREEN, MakeDifSdu(SDUP_CRYPTO,
    MakeDataPdu(BLUE_DIF,ALICE_BLUE,BOB_BLUE,
    MakeDafSdu(SDUP_NONE,CDAP_MESSAGE[])))))) &&
  begin(PduSendEvent(ALICE_BLUE, MakeDataPdu(BLUE_DIF,ALICE_BLUE,BOB_BLUE,
    MakeDafSdu(SDUP_NONE,CDAP_MESSAGE[])))) ->
 end(PduAcceptEvent(BOB_BLUE, MakeDataPdu(BLUE_DIF,ALICE_BLUE,BOB_BLUE,
    MakeDafSdu(SDUP_NONE,CDAP_MESSAGE[]))))
RESULT event(PduAcceptEvent(BOB_BLUE,p)) ==>
 event(PduSendEvent(ALICE_BLUE,p)) is true.
-- Query event(CdapReceiveEvent(BOB_SERVER,CDAP_MESSAGE[])) ==>
 event(CdapSendEvent(ALICE_CLIENT,CDAP_MESSAGE[]))
Completing...
Starting query event(CdapReceiveEvent(BOB_SERVER,CDAP_MESSAGE[])) ==>
 event(CdapSendEvent(ALICE_CLIENT,CDAP_MESSAGE[]))
goal reachable: begin(CdapSendEvent(ALICE_CLIENT,CDAP_MESSAGE[])) ->
 end(CdapReceiveEvent(BOB_SERVER,CDAP_MESSAGE[]))
```

```
RESULT event(CdapReceiveEvent(BOB_SERVER,CDAP_MESSAGE[])) ==>
 event(CdapSendEvent(ALICE_CLIENT,CDAP_MESSAGE[])) is true.
```

In this chapter, we have provided a formal analysis of selected attacks on RINA network hosts and communication. We have applied ProVerif tool to create a formal model of RINA network and selected attacks. Employing capabilities of ProVerif, we were able to prove formally that applying Crypto SDU Protection Policy these attacks can be mitigated. Among the analysed attacks, the replay attack has lead to the most interesting security properties. While the output from ProVerif is easy to understand, for a better explanation of attacks the simulation model using RinaSim has been developed (Figure 32) . The simulations present traces found by the ProVerif that corresponding to the analysed attacks. Simulation output and captured messages that can be analysed using RINASim PduViewer (Figure 33) provide additional details. The scope of the presented analysis is limited by the Dolev-Yao model of the attacker which is implemented by the ProVerif. In this settings, the high-level security properties of RINA architecture can be analysed. For a detailed analysis of SDU protection functions, it would be more suitable to develop a computation model and apply CryptoVerif tool, for instance. The presented approach also demonstrated that formal analysis of security properties of a real network architecture is possible when supported by available formal tools.

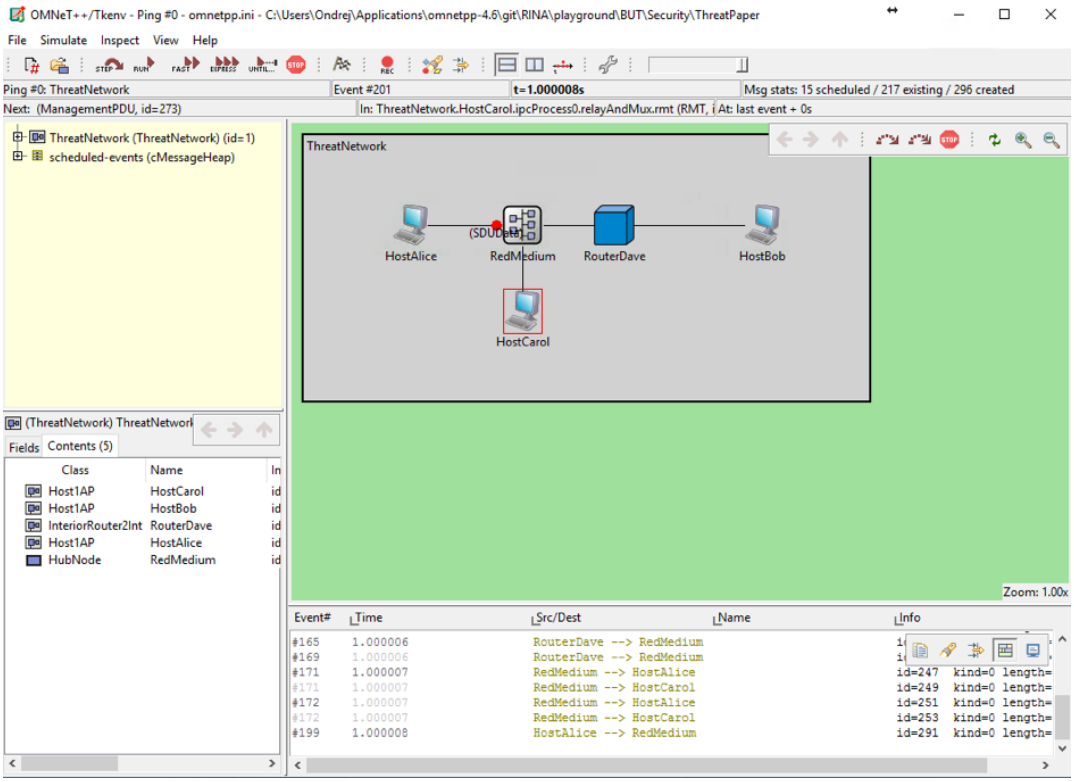

Figure 32. A RINASim Simulation Model

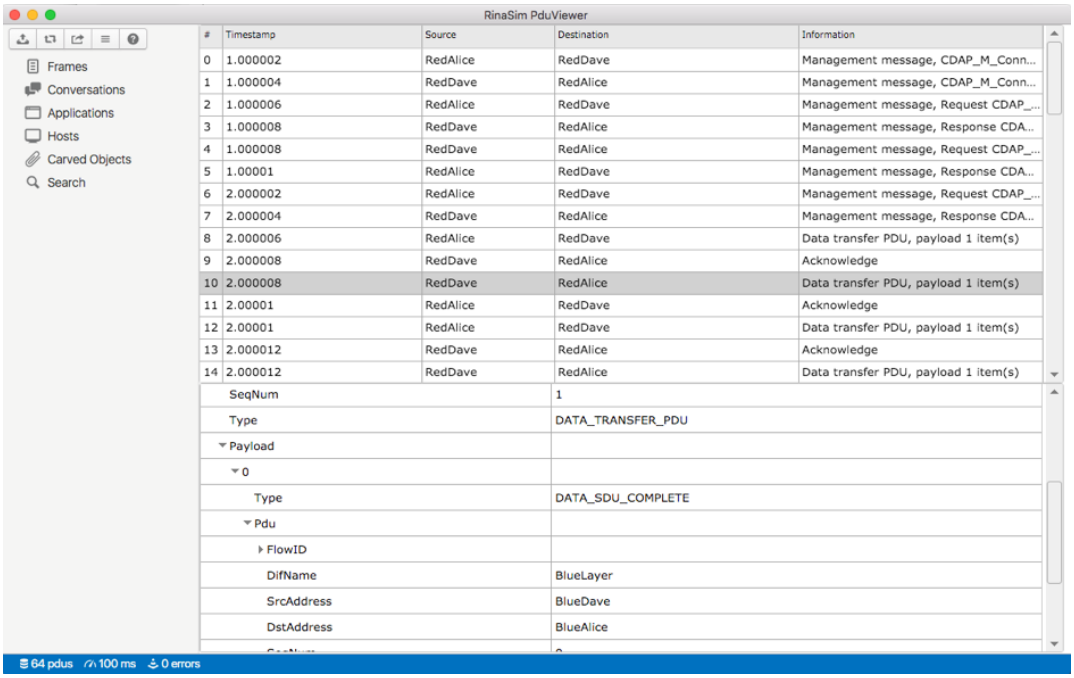

Figure 33. RINASim PDUViewer showing the content of captured PDU

8. Resiliency and High Availability

8.1. Resilient Routing Policies

In this section we report the final version of WP4 research on Loop-Free Alternate mechanism for RINA. In particular, the final specification and implementation on the IRATI prototype are described. Finally, we illustrate the outcomes of a complete experiment involving LFA, carried out with the IRATI stack.

8.1.1. Specification of the Loop Free Alternates (LFA) routing policy

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.

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 */
```

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 34, and its corresponding routing table as calculated by A is shown in Table 8. Note that from A to B there are 2 N-1 flows with different QoS.

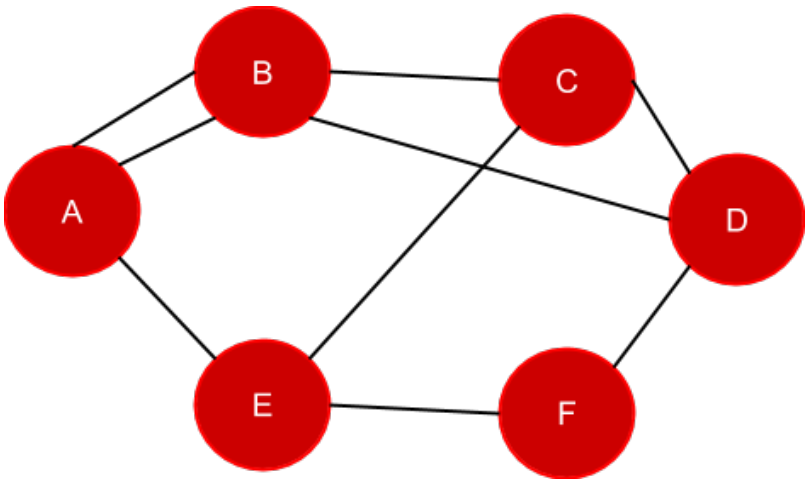

Figure 34. An example connectivity graph

Table 8. 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

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 9. 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.

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 35. 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.

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.

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.

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.

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.

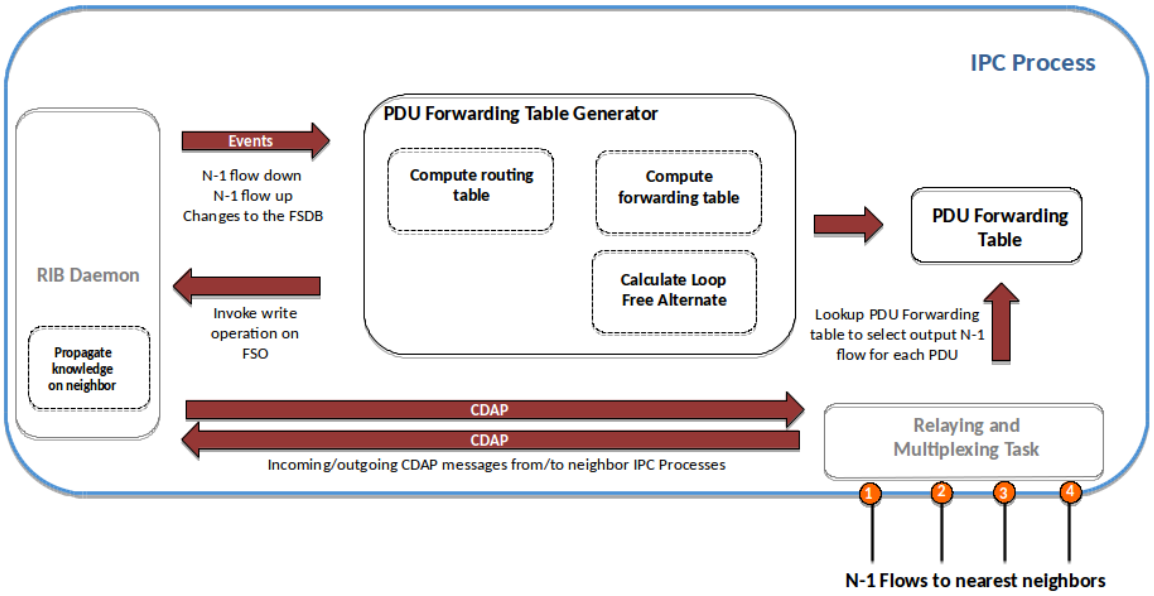

Figure 35. Cooperation of tasks in the IPC process

8.1.2. Implementation in IRATI

The LFA mechanism requires functionalities to be implemented both in user-space and kernel-space.

The initial PoC implementation of the user-space LFA policy has already been presented in section 7.2 of PRISTINE D4.2. That document also introduces some useful terminology. The IPC process on which the routing and LFA computation happens is referred to as **source node**, while the expression **neighbor of a node** refers to another node towards which the first node has a direct link (N-1 flow) in the DIF graph.

After the computation of the regular routing-table by means of the Dijkstra Algorithm, the LFA algorithm is run to search for next-hop alternates. The search for alternates is carried out for all nodes, except the source node, i.e. for all the possible destinations addresses in the routing table of the source node. This is different from what reported in D4.2, where the LFA search was not performed for the neighbors of the source node. This was a design mistake that used to affect very simple topologies, like the following one

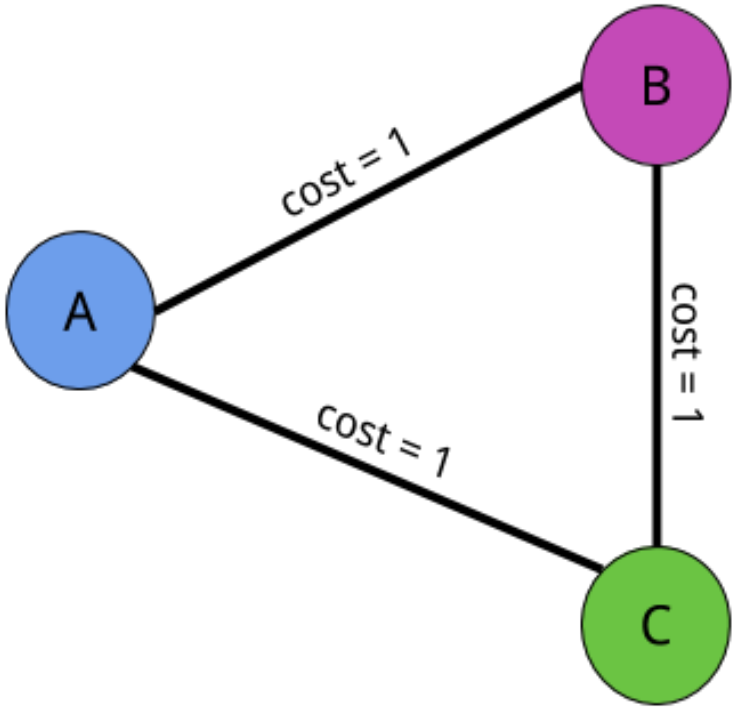

Figure 36. Topology for which original LFA implementation was incorrect

When running the algorithm on node A (figure Figure 36), B would not be selected as LFA next-hop for C (as it should), simply because C is neighbor of A. The user-space implementation has been corrected to fix this problem. The following listing reports the updated pseudocode of the LFA algorithm:

```
src_dist_vec ← computeDistVec(graph, src_node)

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

for each node different from src_node {
  for each 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
    }
  }
}
```

These entries are then pushed down to the PDU Forwarding Function (PFF) policy in kernel space. The LFA PFF contains a table that maps a destination address unto a primary port-id and a list of possible Loop Free Alternates.

The following listing shows what happens when a port-id goes down:

```
add port_id to list of ports that are down

for each entry in routing table {
  if next_hop == port_id {
    for each alt_port_id in alternate port id list {
      if alt_port_id is not down {
        next_hop = alt_port_id;
        break;
      }
    }
  }
}
```

The port-id that is down is added to a list with all the port-ids that are currently down. Next, the routing table is checked to see if there are entries that have the port-id that went down as next hop. If this is the case, an alternate is searched for in the list of possible Loop Free Alternates. If the Loop Free Alternate port-id is not down, the next hop is set to the LFA. If no possible LFA is found, the next hop stays the same, and packets will get dropped until connectivity is restored.

The following listing shows what happens when a port-id becomes available again (e.g. connectivity is restored). Reacting to a port-id can be enabled or disabled on inserting the kernel module.

```
if reacting to port_id up event {

remove port_id from list of ports that are down

for each entry in routing table {
  if next_hop == primary_port_id {
    continue;
  }

  if port_id == primary_port_id {
    next_hop = primary_port_id;
```

```
      continue;
    } else {
      foreach alt_port_id in alternate port id list {
        if alt_port_id is not in list of ports that are down {
          next_hop = alt_port_id;
          break;
        }
      }
    }

}
```

The port-id that is now up again is removed from the list of port-ids that are currently down. Next, each entry in the routing table is checked. If the next hop is still the primary port-id, no action is required. If the port-id that is back up was the primary port-id (not a LFA), then the primary port-id is restored. If the port-id that is back up is a LFA for an entry, and the entry's primary port-id is down, then the LFA is selected as the next hop.

8.1.3. Validation

Validation of the implementation was performed as part of T6.1 and reported in PRISTINE D6.2. In particular, the experiments reported in D6.2 measures the timeliness with which the LFA policy is able to recover from network failure.

In order to verify the correctness of the LFA implementation in a real scenario, however we performed an experiment with the IRATI stack to see the resilient routing in action.

The network configuration for the experiment is shown in the following figure.

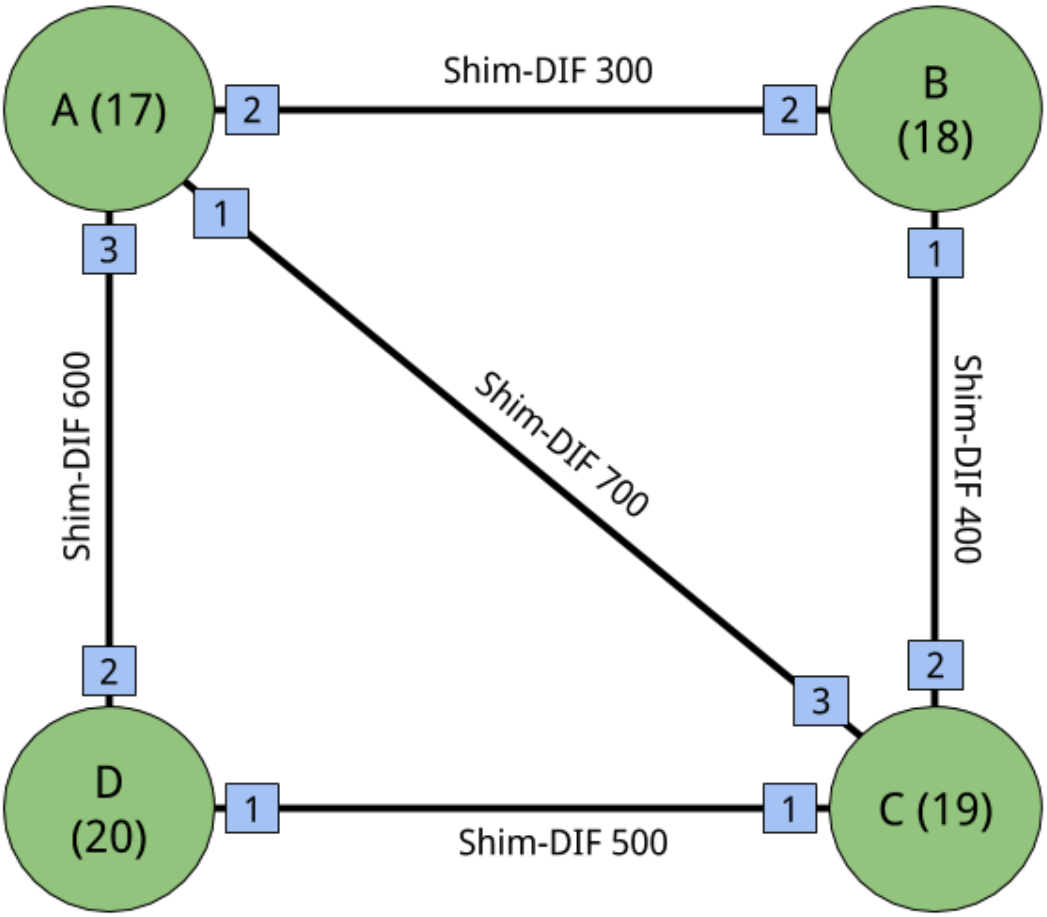

Figure 37. Experiment scenario for LFA fast-reroute

The four nodes - A, B, C and D - are connected in a circle by Ethernet point-to-point links, with an additional link directly connecting nodes A and C. In the figure, the links labels refers to the name of the associated Shim Ethernet over 802.1q, with the name indicating the VLAN id used on the link. A normal DIF spans over the four nodes, providing IPC services to applications by means of the five Shim DIFs. The number right below each node represents the address of the node in the normal DIF.

The purpose of the experiment is to prove that a distributed application exchanging messages between node A and node C is not affected by the failure of links 700 and 600. We expect the network to fast-reroute packets on the next best path between A and C, without causing significant (e.g. more than 30 ms) service interruptions to the distributed application.

Once the enrollment procedures for the normal DIFs are complete, and the link-state algorithm has completely propagated the routing information, the normal DIF will have an N-1 flow for each link (A-B, B-C, C-D, D-A, A-

C). In the figure, the blue squares on the links ends contains the port id of the corresponding N-1 flow.

The log of the IPC Process Daemon on node A shows the computation of the resilient routing table from the Flow State Database and then the computation of the PDU Forwarding Table from the routing table:

```
[...]
358(1464192553)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=17-18
358(1464192553)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=17-19
358(1464192553)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=17-20
358(1464192553)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=18-17
358(1464192553)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=18-19
358(1464192553)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=19-17
358(1464192553)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=19-18
358(1464192553)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=19-20
358(1464192553)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=20-17
358(1464192553)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=20-19
358(1464192553)#ipcp[4].routing-ps-link-state (DBG): Added entry to
 routing table: destination 18, next-hop 18
358(1464192553)#ipcp[4].routing-ps-link-state (DBG): Added entry to
 routing table: destination 19, next-hop 19
358(1464192553)#ipcp[4].routing-ps-link-state (DBG): Added entry to
 routing table: destination 20, next-hop 20
358(1464192553)#ipcp (DBG): Node 19 selected as LFA node towards the
 destination node 18
358(1464192553)#ipcp (DBG): Node 18 selected as LFA node towards the
 destination node 19
358(1464192553)#ipcp (DBG): Node 20 selected as LFA node towards the
 destination node 19
358(1464192553)#ipcp (DBG): Node 19 selected as LFA node towards the
 destination node 20
358(1464192553)#ipcp[4].resource-allocator-ps-default (DBG): Got 3 entries
 in the routing table
```

```
358(1464192553)#ipcp[4].resource-allocator-ps-default (DBG): Processing
 entry for destination 18
358(1464192553)#ipcp[4].resource-allocator-ps-default (DBG): NHOP 18 -->
 N-1 port-id: 2
358(1464192553)#ipcp[4].resource-allocator-ps-default (DBG): NHOP 19 -->
 N-1 port-id: 1
358(1464192553)#ipcp[4].resource-allocator-ps-default (DBG): Processing
 entry for destination 19
358(1464192553)#ipcp[4].resource-allocator-ps-default (DBG): NHOP 19 -->
 N-1 port-id: 1
358(1464192553)#ipcp[4].resource-allocator-ps-default (DBG): NHOP 18 -->
 N-1 port-id: 2
358(1464192553)#ipcp[4].resource-allocator-ps-default (DBG): NHOP 20 -->
 N-1 port-id: 3
358(1464192553)#ipcp[4].resource-allocator-ps-default (DBG): Processing
 entry for destination 20
358(1464192553)#ipcp[4].resource-allocator-ps-default (DBG): NHOP 20 -->
 N-1 port-id: 3
358(1464192553)#ipcp[4].resource-allocator-ps-default (DBG): NHOP 19 -->
 N-1 port-id: 1
[...]
```

As the listing shows, after the computation of the regular routing table the LFA algorithm is run to look for LFA nodes for destinations B, C and D. As expected, node C (identified by address 19) is found to be an LFA for the destination B (identified by node 18). Similarly, nodes B and D are LFA nodes for destination C, and node C is an LFA node for destination D. The LFA nodes are used to extend the routing table entries, so that each destination is associated to multiple next-hop addresses. As an example, the routing table entry for destination C (address 19) has three next-hops: C (19) as the primary one, and B (18) and D (20) as LFA.

Once the extended routing table has been computed, it can be immediately translated into a PDU Forwarding Table, by converting each next-hop address into a next-hop port-id. As an example, the next-hops 19, 18 and 20 routing table entry for destination C are translated into port-ids 1, 2 and 3, respectively. These port-ids are the ones local to node A.

The next listing shows the computation of routing and forwarding tables for node C:

```
[...]
```

```
354(1464192418)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=17-18
354(1464192418)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=17-19
354(1464192418)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=17-20
354(1464192418)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=18-17
354(1464192418)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=18-19
354(1464192418)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=19-17
354(1464192418)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=19-18
354(1464192418)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=19-20
354(1464192418)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=20-17
354(1464192418)#ipcp[4].routing-ps-link-state (DBG): Processing flow state
 object: /resalloc/fsos/key=20-19
354(1464192418)#ipcp[4].routing-ps-link-state (DBG): Added entry to
 routing table: destination 17, next-hop 17
354(1464192418)#ipcp[4].routing-ps-link-state (DBG): Added entry to
 routing table: destination 18, next-hop 18
354(1464192418)#ipcp[4].routing-ps-link-state (DBG): Added entry to
 routing table: destination 20, next-hop 20
354(1464192418)#ipcp (DBG): Node 18 selected as LFA node towards the
 destination node 17
354(1464192418)#ipcp (DBG): Node 20 selected as LFA node towards the
 destination node 17
354(1464192418)#ipcp (DBG): Node 17 selected as LFA node towards the
 destination node 18
354(1464192418)#ipcp (DBG): Node 17 selected as LFA node towards the
 destination node 20
354(1464192418)#ipcp[4].resource-allocator-ps-default (DBG): Got 3 entries
 in the routing table
354(1464192418)#ipcp[4].resource-allocator-ps-default (DBG): Processing
 entry for destination 17
354(1464192418)#ipcp[4].resource-allocator-ps-default (DBG): NHOP 17 -->
 N-1 port-id: 3
354(1464192418)#ipcp[4].resource-allocator-ps-default (DBG): NHOP 18 -->
 N-1 port-id: 2
354(1464192418)#ipcp[4].resource-allocator-ps-default (DBG): NHOP 20 -->
 N-1 port-id: 1
354(1464192418)#ipcp[4].resource-allocator-ps-default (DBG): Processing
 entry for destination 18
```

```
354(1464192418)#ipcp[4].resource-allocator-ps-default (DBG): NHOP 18 -->
 N-1 port-id: 2
354(1464192418)#ipcp[4].resource-allocator-ps-default (DBG): NHOP 17 -->
 N-1 port-id: 3
354(1464192418)#ipcp[4].resource-allocator-ps-default (DBG): Processing
 entry for destination 20
354(1464192418)#ipcp[4].resource-allocator-ps-default (DBG): NHOP 20 -->
 N-1 port-id: 1
354(1464192418)#ipcp[4].resource-allocator-ps-default (DBG): NHOP 17 -->
 N-1 port-id: 3
[...]
```

The output is specular to the one illustrated for node A. As an example, destination A is reachable through the primary next-hop A, and alternate next-hops B and D. In terms of port-ids, the primary next-hop is accessed through C-local port-id 3, while B and D are accessed through port-ids 2 and 1, respectively.

In this initial situation, we start the rina-echo-time tool in server mode on node C:

```
[...]
$ rina-echo-time -l
[...]
```

and we start the same tool in client mode on node A, instructing it to send 100 packets per-second for a 1000 seconds:

```
[...]
$ rina-echo-time -w 10 -c 100000
[...]
```

In the initial situation all the traffic flows through the Shim DIF 700, which is the shortest path between nodes A and C. This was verified using a traffic monitoring/sniffing tool (tcpdump) on nodes A and C. As soon as the link of Shim 700 goes down (emulated by disabling A and C network interfaces on link A-C), the LFA fast-rerouting mechanisms is triggered, as reported in the kernel log of node A:

```
[...]
[   47.972149] rina-shim-eth-vlan(INFO): Device ens6.700 goes down
```

```
[   47.972661] rina-normal-ipc(INFO): N-1 flow with pid 1 went down
[   47.973810] rina-pff-lfa(INFO): Port-id 1 goes down
[   47.974853] rina-pff-lfa(INFO): Looking for an alternate port-id for
 destination address 19
[   47.976160] rina-pff-lfa(INFO): Found alternate port-id 3
[...]
```

The log shows that port-id 1 (corresponding to Shim 700) goes down and it is replaced by the alternate port-id 3, which corresponds to Shim DIF 600. A similar information is reported in the kernel log of node C:

```
[...]
[   80.656550] rina-shim-eth-vlan(INFO): Device ens6.700 goes down
[   80.657077] rina-normal-ipc(INFO): N-1 flow with pid 3 went down
[   80.657583] rina-pff-lfa(INFO): Port-id 3 goes down
[   80.658005] rina-pff-lfa(INFO): Looking for an alternate port-id for
 destination address 17
[   80.658701] rina-pff-lfa(INFO): Found alternate port-id 1
[...]
```

Therefore, the failure of link A-C results into the traffic between A and C being rerouted through node D. The ping application does not even notice the change, since the application flow is not deallocated, and the RMT of the the normal IPCPs on nodes A and C are still able to route the packets.

In this experiment it is also possible to validate the case of multiple failures. As soon as node D goes down (emulated by disabling A interface on link A-D and C interface on link D-C), the LFA fast-rerouting is triggered again. The kernel log on A reports the following:

```
[...]
[   62.842511] rina-shim-eth-vlan(INFO): Device ens5.600 goes down
[   62.843812] rina-normal-ipc(INFO): N-1 flow with pid 3 went down
[   62.845199] rina-pff-lfa(INFO): Port-id 3 goes down
[   62.846257] rina-pff-lfa(INFO): Looking for an alternate port-id for
 destination address 20
[   62.847546] rina-pff-lfa(INFO): No alternate port-id is available
[   62.848639] rina-pff-lfa(INFO): Looking for an alternate port-id for
 destination address 19
[   62.850963] rina-pff-lfa(INFO): Found alternate port-id 2
[...]
```

The log shows that port-id 3 (corresponding to Shim 600) goes down and it is replaced with the alternate port-id 2, which corresponds to Shim 300. In addition to that, the log shows that node D (whose address is 20) is no longer reachable, as expected.

Similarly, the kernel log on node C shows that port-id 1 (Shim 500) goes down, and it is replaced by the alternate port-id 2, which corresponds to Shim 400:

```
[...]
[   84.436484] rina-shim-eth-vlan(INFO): Device ens5.500 goes down
[   84.439931] rina-normal-ipc(INFO): N-1 flow with pid 1 went down
[   84.443698] rina-pff-lfa(INFO): Port-id 1 goes down
[   84.446090] rina-pff-lfa(INFO): Looking for an alternate port-id for
 destination address 20
[   84.448857] rina-pff-lfa(INFO): No alternate port-id is available
[   84.450787] rina-pff-lfa(INFO): Looking for an alternate port-id for
 destination address 17
[   84.454592] rina-pff-lfa(INFO): Found alternate port-id 2
[...]
```

As a consequence the failure of node D results in the traffic being re-routed through node B. Once again, the ping application is not affected significantly, as packets keeps flowing between A and C.

8.1.4. Conclusions

The Loop Free Alternate technique has been studied and implemented in order to improve the resiliency of RINA networks. Differently from what happens in TCP/IP, the LFA routing policy can be used at any layer: in lower layers to cope with the failure of transmission technologies (e.g. Ethernet, WiFI); in higher layers to react to broader failures, or simply rearrangements in the lower layer DIFs.

The experiments conducted with the IRATI stack - also reported in D6.2 - have shown that the time to recovery from a link failure can be as small as ~2 milliseconds, which is about two orders of magnitude less then the time required for the routing algorithm to converge again.

8.2. Management of names in Networking

8.2.1. Introduction

This section describes the use of names in networking, with a focus on the Recursive InterNetwork Architecture (RINA). We propose name-space management in RINA to be performed at 4 different levels:

- Within each application
- Within each processing system
- Within each Distributed Application Facility (DAF)
- And global name-space management

We believe this will make name-space management in RINA scale. This section is part of the resiliency section of this document since it is required to explain the use of the leveraging of the whatever-cast mechanism for resiliency purposes. A whatever-cast name is just a name that can be resolved like any other name in RINA. We start by introducing some well-defined definitions to avoid as much ambiguity as possible. Next we describe the different parts that an application can be identified with. Lastly we describe the different levels of name-space management and their service definitions.

8.2.2. Glossary [6]

Bind - to choose a specific lower-level implementation for a particular higher-level semantic construct. In the case of names, binding is choosing a mapping from a name to a particular object, usually identified by a lower-level name.

Directory - an object consisting of a table of bindings between symbolic names and objects. A directory is an example of a name-space

Limited name-space - a name-space in which only a few names can be expressed, and therefore names must be reused.

Name - in practice, a character- or bit-string identifier that is used to refer to an object on which computation is performed. Abstractly, an element of a name-space

[6] Definitions based on those from “Naming and Binding of objects” by J. H. Saltzer 1978

Name-space - a particular set of bindings of names to objects: a name is always interpreted relative to some name-space

Naming hierarchy - a naming network that is constrained to a tree-structured form.

Naming network - a directory system in which a directory may contain the name of any object, including another directory. An object is located by a multi-component path name relative to some working name-space

Object - a software (or hardware) structure that is considered to be worthy of a distinct name.

Path name - a multiple component name of an object in a naming network. Successive components of the path name are used to select entries in successive directories. The entry selected is taken as the directory for use with the next component of the path name. For a given starting directory, a given path name selects at most one object from the hierarchy.

Reference name - the name used by one object (e.g., a program) to refer to another object.

Resolve - to locate an object in a particular name-space, given its name.

Root - the starting directory of a naming hierarchy.

Search - abstractly, to examine several name-spaces looking for one that can successfully resolve a name. In practice, the systematic examination of several directories of a naming network, looking for an entry that matches a reference name presented by some application.

Synonym - one of the multiple names for a single object permitted by some directory implementations.

Tree name - a multiple component name of an object in a naming hierarchy. The first component name is used to select an entry from a root directory, which selected entry is used as the next directory. Successive components of the tree name are used for selection in successively selected directories. A given tree name selects at most one object from the hierarchy.

Unique identifier - a name, associated with an object at its creation, that differs from the corresponding name of every other object that has ever been created by a processing system.

Unlimited name-space - a name-space in which names never have to be reused.

Working directory - in a naming network, a directory relative to which a particular path name is expressed.

8.2.3. Applications in RINA

In RINA Applications are identified with

- The application process name: The name of the application
- The application process instance: An application needs to be instantiated. This is to uniquely identify which instance of the application it is, local to the processing system.
- The application entity name: An application may consist of different application entities that may employ a different protocol. Communication is always between two entities.
- The application entity instance: The instantiation identifier of the entity, local to the application.

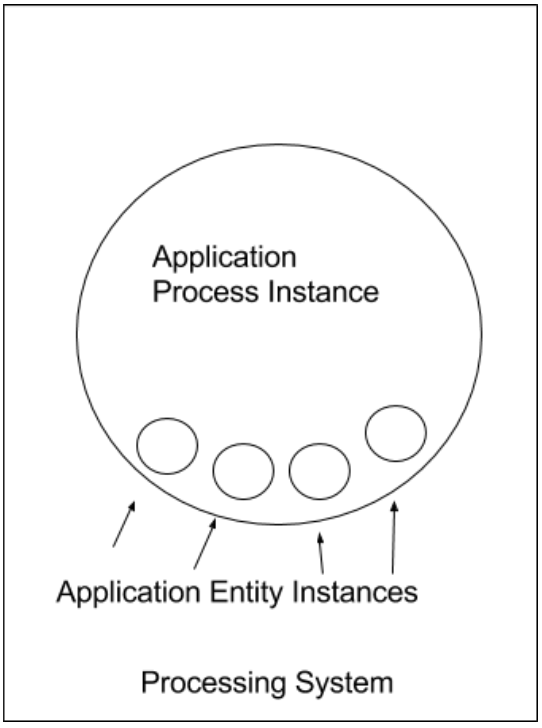

Figure 38. An application within a processing system.

8.2.4. Application Name-space Management

Within the application, the application name is known and the application entity instances with their respective application entity names. The name-space management within the application consists of managing the application entities. Flow are established between application entity instances. The application name-space management has the following service definition:

resolveName.submit(application-entity)

When invoked

This is invoked when a flow allocation request arrives for the application, and the application has to find the right entity instance.

Action upon receipt

Upon receipt, if the application entity name is found in the application's directory, a list of application entity instances is returned. If no such name was bound, nothing is returned.

registerName.submit(application-entity, object)

When invoked

This is invoked when a new application entity instance needs to be registered.

Action upon receipt

Upon receipt, the application entity name is added to the directory of the application. The reference name of the object is added to the mapping. Note that if there was already an application entity registered with that name, both reference names are kept. Adding the name to the directory does not demand uniqueness.

unregisterName.submit(application-entity, object)

When invoked

This is invoked when an application entity instance needs to be unregistered.

Action upon receipt

Upon receipt, the mapping of application entity name to the reference name of the object is removed. The application entity name is removed from the directory of the application if it was the last instance.

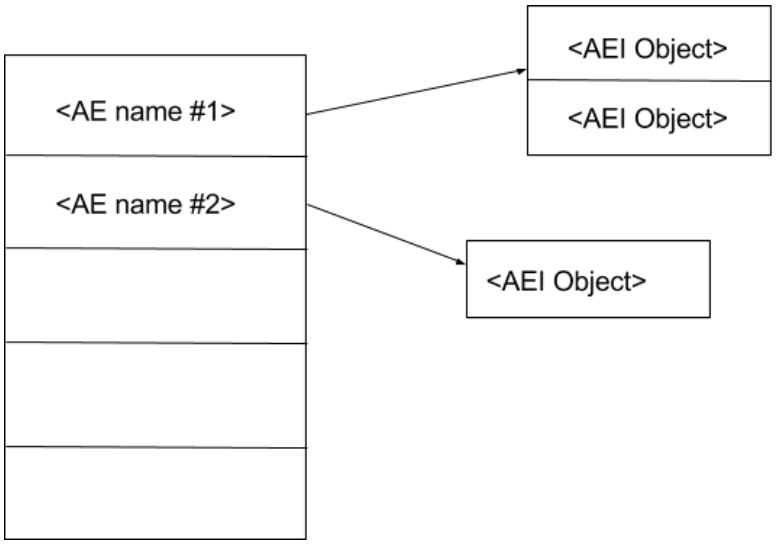

Figure 39. Namespace management within an application process.

8.2.5. Processing System Name-space Management

Within the processing system, application names are kept and their instances. The name space management within the processing system consists of managing the application instances. The processing system name-space management has the following service definition:

resolveName.submit(application-name)

When invoked

This is invoked when a new flow allocation request arrives for an application.

Action upon receipt

If the application is found in the processing system's directory, a list of application instances is returned.

registerName.submit(application-name, object)

When invoked

This is invoked when a new application name is registered with the processing system.

Action upon receipt

The application name is added to the processing system's directory if it was not there yet. The reference name of the object is added to the mapping.

unregisterName.submit(application-name, object)

When invoked

This is invoked when an application name is unregistered by the IPC Resource Manager of the processing system.

Action upon receipt

The mapping of the application name to the reference name of the object is removed. The application name is removed from the processing system's directory if it was the last one that was registered with that name.

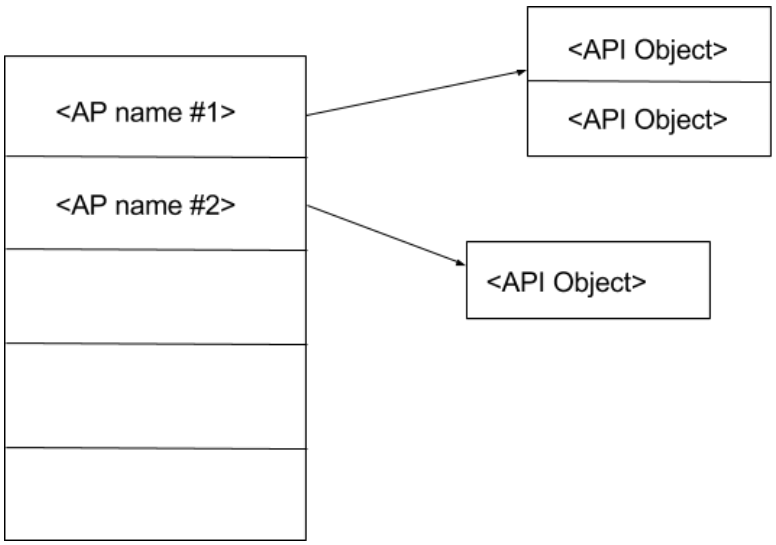

Figure 40. Namespace management within a processing system.

8.2.6. Distributed Application Facility Name-space Management

Within a Distributed Application Facility (DAF), application names are kept. This means that it is kept within the Resource Information Base (RIB) of every DAF member, e.g. every Distributed Application Process in the DAF. The name space management of the DAF is concerned with managing application names and their mapping on location dependant names (often

referred to as addresses) in the DAF. The DAF name-space management has the following service definition:

resolveName.submit(application-name)

When invoked

This is invoked when an application name is to be resolved to a list of addresses.

Action upon receipt

If the application name is found in the DAF's directory, a list of addresses is returned.

registerName.submit(application-name, address)

When invoked

This is invoked when a new application name is registered in the DAF.

Action upon receipt

The application is added to the DAF's directory if it was not there yet. The address is added to the list of addresses in the mapping.

unregisterName.submit(application-name, address)

When invoked

This is invoked when an application name is unregistered from the DAF.

Action upon receipt

The mapping of the application name to the address is removed from the DAF's directory. The application name is removed from the DAF's directory if it was the last one that was registered with that name.

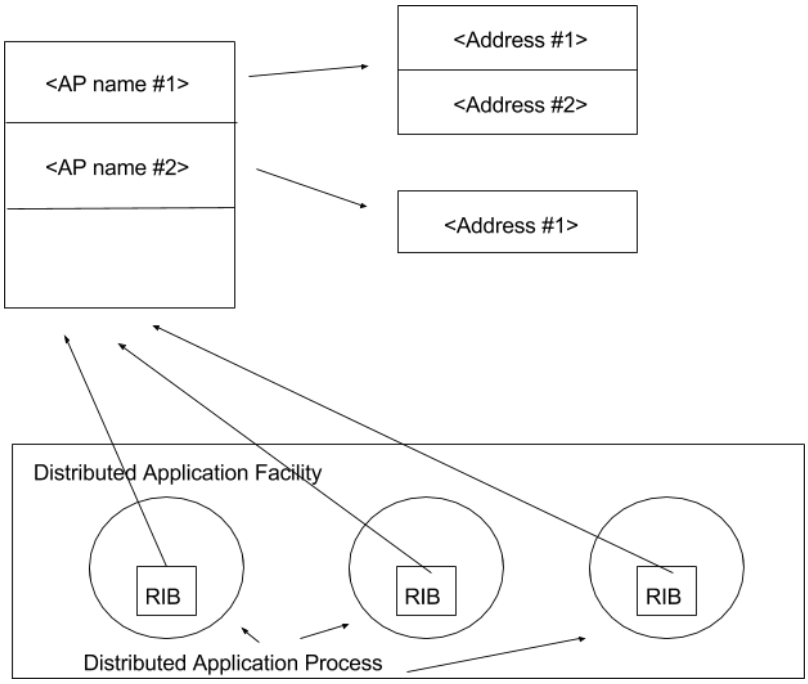

Figure 41. Namespace management within a Distributed Application Facility.

8.2.7. Root Name-space Management

Within the root name-space management, application names are kept. The root name-space management is a DAF that holds the mapping of application names to DAF name. Note that a DAF name is also just an application name; it's just a distributed application. There can be multiple root name-spaces

resolveName.submit(application-name, daf-names)

When invoked

This is invoked when an application name needs to be resolved to a list of DAF names.

Action upon receipt

Upon receipt, if the application can be found in the directory, a list of DAF names is returned.

registerName.submit(application-name, daf-names)

When invoked

This is invoked when a new application is registered in a DAF.

Action upon receipt

The application is added to the root directory if it was not there yet. The DAF name is added to the list of DAF names in the mapping.

unregisterName.submit(application-name, daf-names)

When invoked

This is invoked when an application is unregistered from the DAF.

Action upon receipt

The mapping of the application name to the DAF name is removed from the root directory. The application name is removed from the directory if it was the last one that was registered with that name.

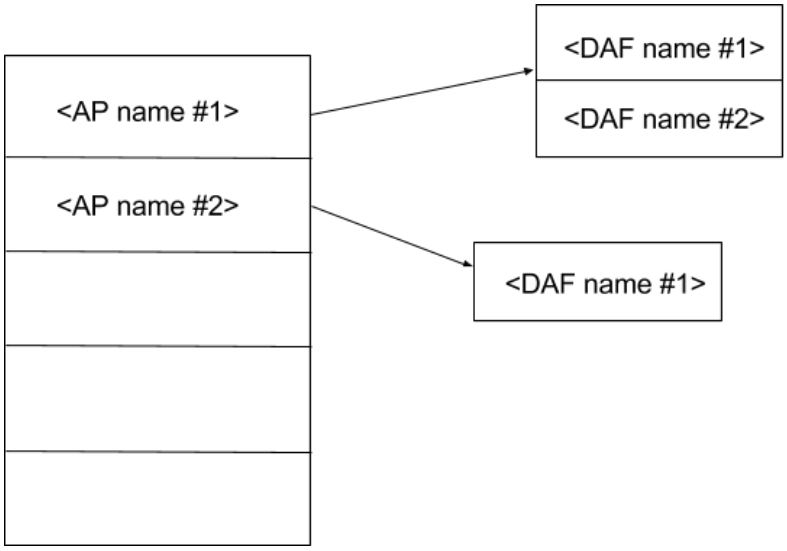

Figure 42. Global name-space management.

8.3. Providing Resiliency through Whatever-cast

This section describes a technique for providing resiliency in the Recursive InterNetwork Architecture. Resiliency can be achieved by leveraging on one of the mechanisms of the architecture: the concept of whatever-cast

Whatever-cast is a generalization of unicast, any-cast, multicast and broadcast. Whatever-cast names consist of a non-empty set and a rule. Resolving the name invokes the rule, which may return one or many elements of the set. How the rule is defined and what are its inputs depends on the specific application.

A whatever-cast name can be used to identify the destination (or target) application in a flow allocation request, where it can be resolved in different ways, depending on the set and rule:

1. When the set contains just an element, the resolution rule must return that element. This corresponds to traditional unicast.
2. When the set contains more than one element and rule returns a single element, we have any-cast
3. When the set contains more than one element and the rule returns more than one element, there are two sub-cases, depending on whether the requesting application is a member of the set or not. This is referred to as multicast.

In case 3, when the requester is not a member of the set, we can distinguish three additional sub-cases, depending on the allowed information flow between requester to the destination:

1. Write-only, where the requester is only allowed to send information to the destination. Traditional multicast subscriptions, e.g. a streaming audio/video service, are common examples where the rule returns all the members in the set. In these cases, the streaming server is the requester and the set of subscribers are the members of the set.
2. Read-only, where the requester is only allowed to receive information from the destination, without the ability to send any response to the destination. As an example, a monitoring application may be receiving information from a sensor network. In this case the monitoring application is the requester and the sensors are the members of the set, with the rule returning all of them.
3. Read-write, where the information can flow in both directions.

If the requester is a member of the set, a write by any member goes to all other members returned by the rule. This form has been referred to as multi-peer, in that there are multiple senders.

8.3.1. Proposed whatever-cast scheme

In order to provide resiliency we will use whatever-cast names satisfying the following properties:

- The set contains more than one element with the rule resolving to more than one member of the set.
- The flow requester is not a member of the set.
- Read-write information flow allowed between requester and target.

In the reference scenario illustrated below, an application AppA running on the processing system A wants to establish a flow with a remote application AppB running on the processing system B.

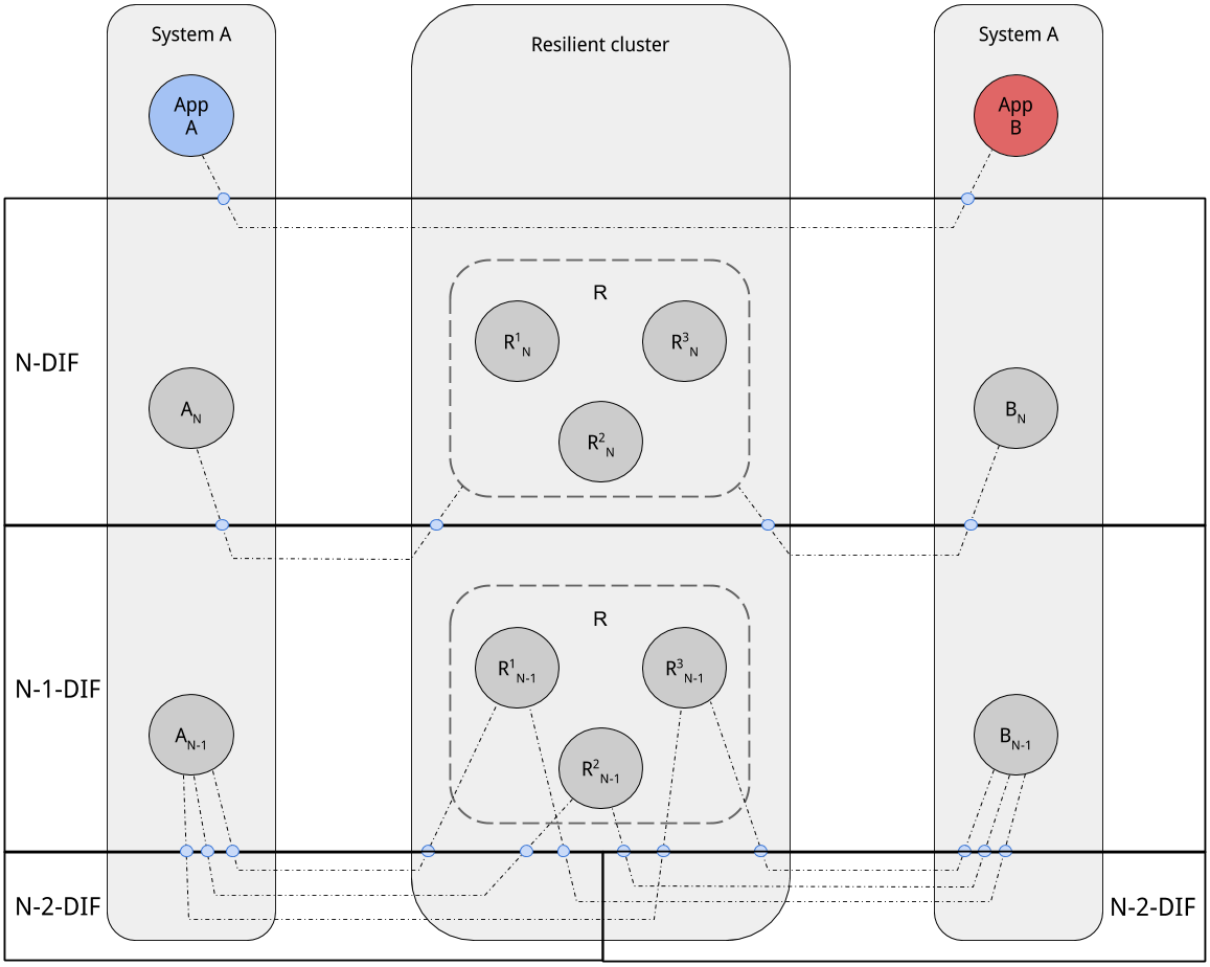

Figure 43. Reference scenario for whatever-cast resiliency

Definitions

1. AppA is an application running in the processing system A.
2. AppB is an application running in the processing system B.
3. A_N and A_{N-1} are IPC processes running in the processing system A.
4. B_N and B_{N-1} are IPC processes running in the processing system A.
5. R^1_N and R^1_{N-1} are IPC processes running in the processing system R^1.

6. R^2_N and R^2_{N-1} are IPC processes running in the processing system R^2.
7. R^3_N and R^3_{N-1} are IPC processes running in the processing system R^3.
8. R^1, R^2 and R^3, are three processing systems forming a resilient cluster.
9. R is a whatever-cast name referencing the resilient cluster at the N-DIF level.
10. p(X, Y) is the port id used by an application (i.e. an IPC Process) X to communicate with the remote application Y.
11. a(X) is the address of IPC Process X in the corresponding DIF.
12. a_N(R) is the address of the whatever-cast set R in the N-DIF.
13. a_{N-1}(R) is the address of the whatever-cast set R in the N-1-DIF.

High level description

When AppA allocates a flow to AppB, a certain amount of resiliency is expected from the network.

Assumptions

1. Applications AppA and AppB allocate a flow by means of the N-DIF.
2. A_N, R^1_N, R^2_N, R^3_N and B_N are registered to the N-1-DIF, and they are enrolled in the N-DIF by means of the N-1-DIF.
3. Both R^1_N, R^2_N, R^3_N have registered the whatever-cast name R to the N-1-DIF. As a consequence, the N-1-DIF is aware of R.
4. A_{N-1}, R^1_{N-1}, R^2_{N-1}, R^3_{N-1} and B_{N-1} are registered to an N-2-DIF, and they are enrolled in the N-1-DIF by means of the N-2 DIF.
5. In the general case, A_{N-1}, R^1_{N-1}, R^2_{N-1}, R^3_{N-1} have an N-2-DIF in common, B_{N-1}, R^1_{N-1}, R^2_{N-1}, R^3_{N-1} have an N-2-DIF in common, but A_{N-1}, and B_{N-1} do not have any N-2-DIF in common.

Workflow of the whatever-cast flow allocation between A_N and R

1. A_N asks for the allocation of a flow towards R.
2. The local Name Space Management of system A figures out that the flow can be allocated through the N-1-DIF, and therefore through the A_{N-1} local IPC process.
3. A_{N-1} performs a lookup for R in its Directory Forwarding Table (DFT) to figure out the address of the remote IPC process to which the CDAP Flow Allocation message should be sent.

4. The lookup triggers the resolution of the whatever-cast name R, which results into a tuple of addresses: $a(R^1_{N-1})$, $a(R^2_{N-1})$, $a(R^3_{N-1})$.
5. A_{N-1} forwards the CDAP Flow Allocation message to both R^1_{N-1}, R^2_{N-1} and R^3_{N-1}.
6. If A_{N-1} collects three positive responses, the flow allocation is successful. If at least two positive responses are received, the flow allocation is considered failed.
7. If the flow allocation is successful, assuming three positive CDAP responses are received. A_{N-1} updates its PDU Forwarding Table (PDUFT) with the following entry $a_{N-1}(R) \rightarrow p(A_{N-1}, R^1_{N-1}), p(A_{N-1}, R^2_{N-1}), p(A_{N-1}, R^2_{N-1})$, which means that every PDU that with destination address $a_{N-1}(R)$ will be forwarded to both R^1_{N-1}, R^2_{N-1}, R^3_{N-1} (multicast forwarding). This will happen for the SDUs sent by AN on the whatever-cast flow just allocated. Clearly, if less positive CDAP responses are received, only the corresponding neighbours will be used in the PDUFT.
8. A_N gets notified about the result of the flow allocation procedure. A whatever-cast N-1-flow now exists between A_N and R_N, with port id $p(A_N, R)$

A similar procedure applies for the allocation of a flow between BN and R.

Observations

After completing the flow allocation procedure between A_N and R, the routing in the N-DIF will then update the PDUFT of A_N with the entry $a(B_N) \rightarrow p(A_N, R)$ which can be used to forward PDUs corresponding to SDUs sent by AppA on the N-flow towards AppB. From the point of view of the N-DIF, each PDU belonging to the flow between AppA and AppB is forwarded by A_N or B_N through the intermediate node R, in unicast. Actually, such an intermediate node is an abstraction to represent a cluster of three elements, and the N-1-DIF performs multicast replication in a transparent way with reference to the N-DIF.

In more detail, a PDU sent by A_N will be received by the RMT of both R^1_N, R^2_N and R^3_N. Each RMT will forward the PDU to BN because of the routing in the N-DIF, using a regular unicast N-1-flow between R^i_N and B_N (not depicted in the figure).

Since PDUs are replicated, both A_N and B_N can receive duplicated PDUs on behalf of AppA or AppB, respectively. Depending of the QoS negotiated

between AppA and AppB through the N-DIF, these duplicated PDUs may be transparently eliminated by the EFCP protocol running in A_N and B_N.

From a management point of view, note that A_{N-1} and B_{N-1} need to know about R - which is a whatever-cast address and how it is going to be resolved. This is needed when A_{N-1} and B_{N-1} look up R in their DFT.

The members of the whatever-cast set should adhere to requirements in terms of latency, hop count in the N-1 DIF, … in order to fulfil the requirements of the offered QoS-cubes.

8.3.2. Detailed working in every layer

The Application Process

The application process can request a certain amount of resiliency from the network through the IPC API. When it asks to allocate a flow to a destination process it should be able to select a QoS-cube with a certain amount of resiliency. A parameter of the QoS-cube might be specifying the number of nines for availability.

Availability	Downtime per year
90 (class 1)	36.5 days
99 (class 2)	3.65 days
99.9 (class 3)	8.76 hours
99.99 (class 4)	52.56 minutes

A different parameter may be the maximum allowable interruption time. This metric sets an upper bound to the time it may take to perform a recovery action. By requesting these parameters from the network, only DIFs that can guarantee this QoS are used to provide IPC to the application.

The N-DIF

In the N-DIF a distinction can be made between IPC Processes:

- Normal IPCPs: These IPCPs provide IPC services to the layer above to applications that request them
- Whatever-cast IPCPs: These are Normal IPCPs that are resilient. Each whatever-cast IPCP has joined a whatever-cast set, with every set containing more than 1 member, ensuring their resilience in case of failure.

Every Normal IPCP in this DIF needs to allocate a flow with at least one whatever-cast IPCP. The flow allocation to a whatever-cast IPCP is similar to regular flow allocation. Instead of a unicast name, a whatever-cast name is however specified as the destination application. Or put a different way, regular flow allocation is a degenerate case of whatever-cast flow allocation.

The network administrator decides which IPCPs will become Whatever-cast IPCPs after they have joined the DIF. After enrollment, these IPCPs are told to join a certain whatever-cast set. This whatever-cast set is also identified by an address so that it can be used in routing. The IPCP sends a CDAP message to its neighbours with a request to join the set. This information is then shared with every member in the DIF. After the IPCP has joined the whatever-cast set it also notifies the N-1 IPCP it is using to communicate with other members of the N-DIF of this fact. It registers the whatever-cast name with the N-1 DIF.

The N-1 DIF

Upon a flow allocation request to a whatever-cast destination, the N-1 IPCP will apply the rule to the set of whatever-cast members to resolve a multicast group. The amount of members that will be resolved by the rule can be all of them, or can be a preconfigured number. The N-1 IPCP will sent a flow allocation request to all of these members. Once at least two flow requests are successful, the whatever-cast flow will be allocated.

It will also notify the IPCPs of the whatever-cast address that will be used. This address is the same as any unicast address being used in the DIF, it is just an address that is not in use at that time. A routing entry for the whatever-cast address is added in the routing table.

When the N-1 IPCP receives an SDU that is destined for the whatever-cast IPCP, it will lookup the whatever-cast address in its forwarding table, to receive the port-ids from the flows that were previously allocated.

8.3.3. Implementation in RINASim

Modular structure of RINASim based on the split of policies and mechanisms allowed us to implement above-mentioned whatever-cast properties quiet easily. This section outlines design changes to RINASim concepts.

Whatever-cast support in QoSCubes

We have updated both QoSCube and QoS requirements abstract data structures to contain the level of resiliency information. We have simply added another signed integer variable called `resiliencyFactor`, which can hold domain specific such as maximum number of allowed downtime during measured period, high-availability of service measured in a number of 9s, etc. Nevertheless, two values are reserved: 0 as default value and -1 as for do not care scenarios, where resiliency is not considered as requirement. Currently implemented policies translating QoS requirement onto available QoSCube consider lower value as better. Application Entity may then specify what resiliency it requires from the network and underlying DIFs. Figure below shows illustrative usage within executed simulation.

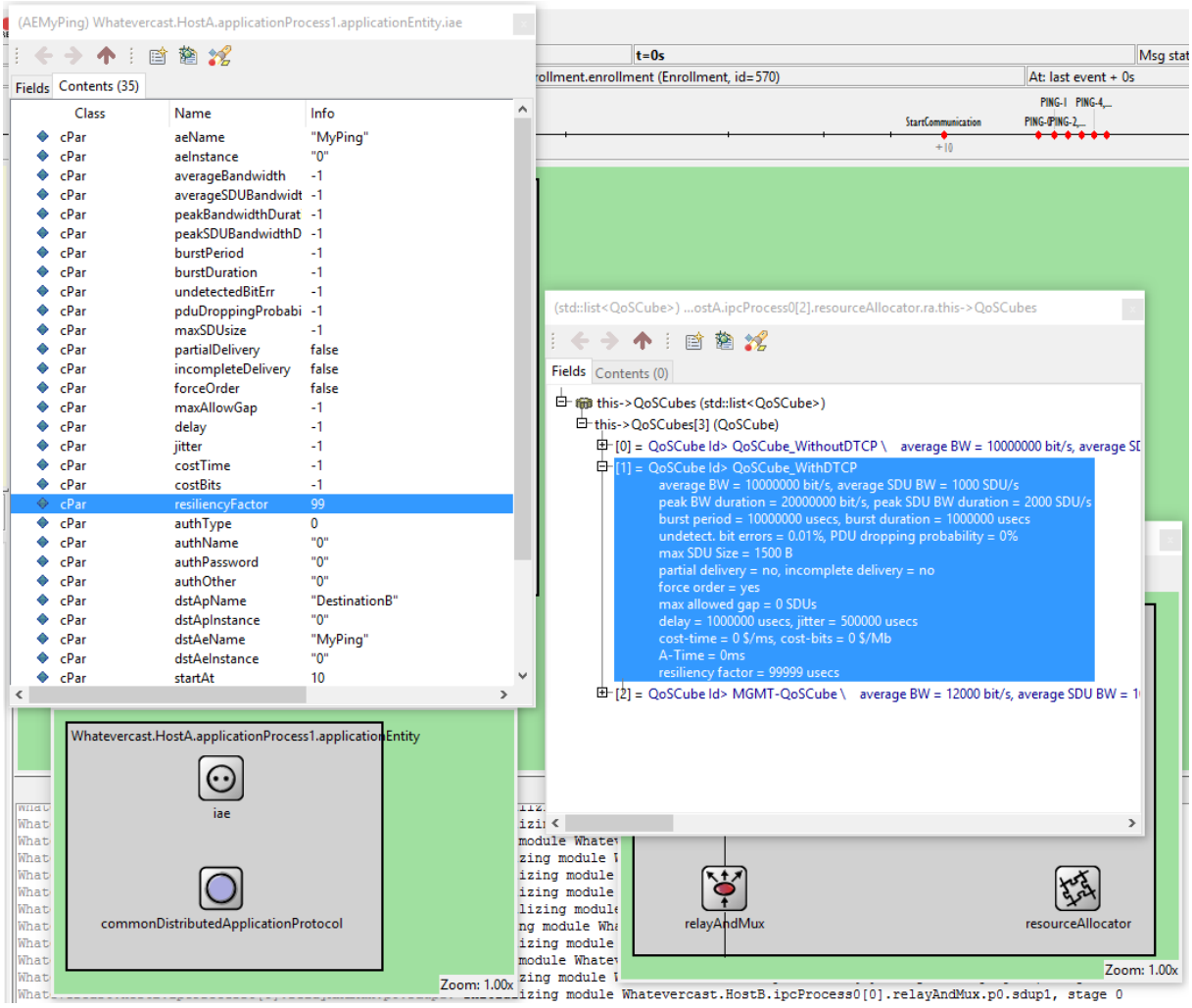

Figure 44. Resiliency Factor parameter

Whatever-cast support in DIF Allocator

Prior to this feature introduction, RINASim was capable to resolve destination Application Process Name (APN) onto first available APN specifying IPC Process (IPCP) which may be used to reach destination APN. We have updated DIF Allocator module implementation to retrieve the list instead of single APN. Hence, DA's Directory is capable to appropriately deal with multiple (N)-to-(N-1) mappings such as the one depicted on figure bellow, where APN *HB_ CommonLayer* can be reached via three IPCPs *hb_ MediumLayerB0*, *hb_ MediumLayerB1* and *hb_ MediumLayerB2*. Policy based approach may be used to choose a subset of retrieved mappings in order to perform unicast, broadcast, any-cast or multicast transfer behaviour

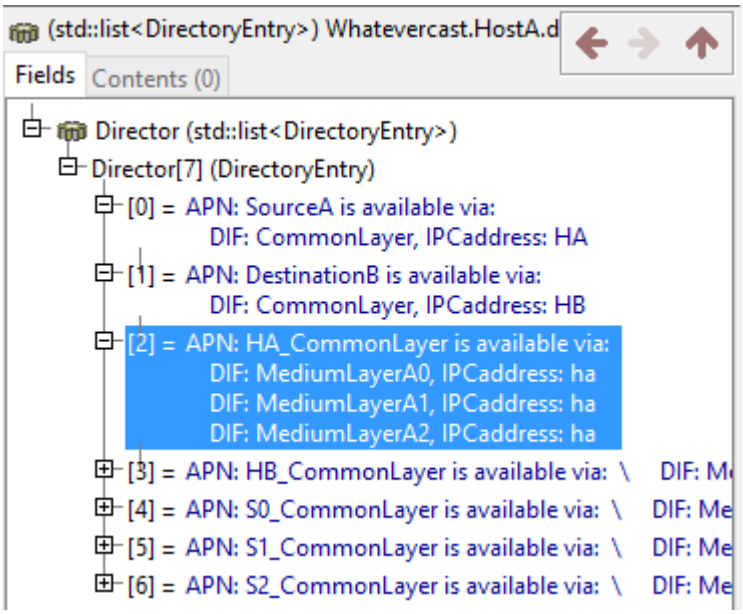

Figure 45. Directory illustration

Whatever-cast support in Flow Allocator

Flow Allocator has been updated to perform multiple flow allocation whenever more than one neighbouring APN leading towards destination is available and resiliency of communication has been specified in QoS requirements Information about available neighbours are stored in DIF Allocator's Neighbour Table (NT). Figure below shows example where IPCP *HB_ CommonLayer* is reachable via three directly connected neighbours *S0_ CommonLayer*, *S1_ CommonLayer* and *S2_ CommonLayer*. Currently, NTs content is statically preconfigured using external XML file

prior to simulation start-up However, we plan to address this limitation in the next RINASim release.

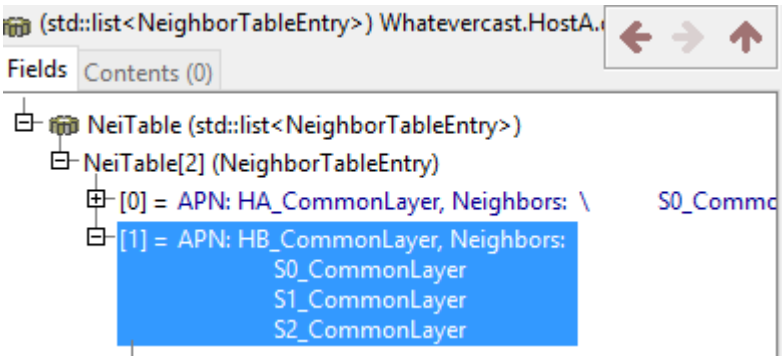

Figure 46. Neighbour table illustration

Demonstrative scenario

Illustrative simulation follows use-case described above. When sending data to whatever-cast name, multiple flows are allocated between HostA and HostB. Each flow goes through different interior router (depicted as Switch0, Switch1 and Switch2). Each flow separately carries a copy of data thus providing resiliency data transfer

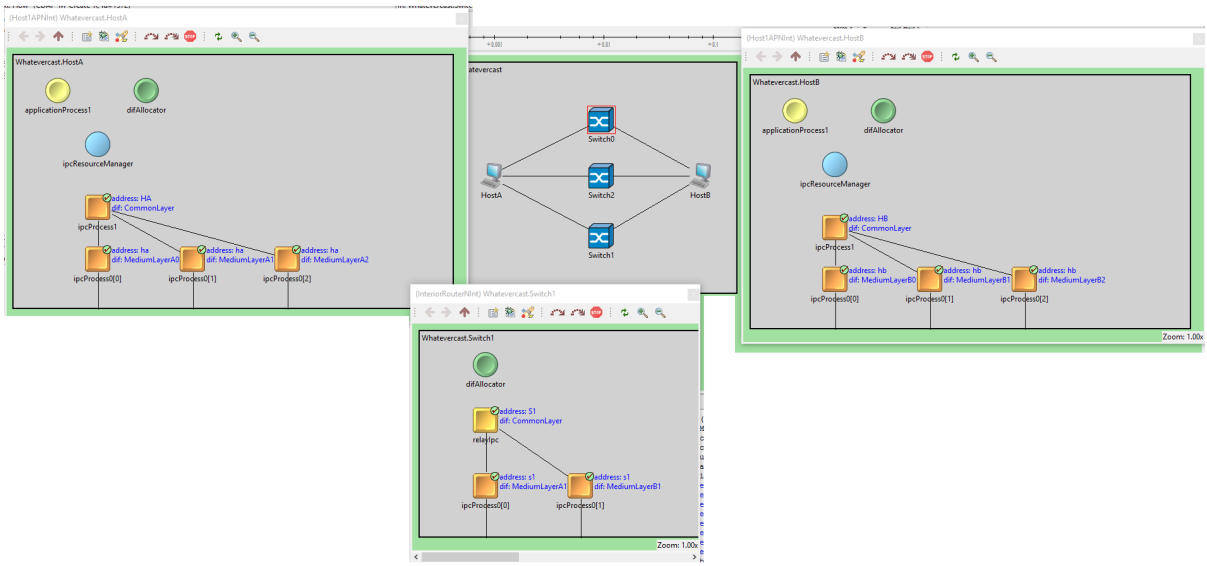

Figure 47. Whatever-cast RINASim topology

Simulation goes through following phases:

1. As the part of simulation setup, HostA and HostB IPCPs are enrolled onto appropriate DIFs (*CommonLayer*, *hb_MediumLayerA0*, *hb_MediumLayerA1*, *hb_MediumLayerA2*,

hb_MediumLayerB0, *hb_MediumLayerB1* and *hb_MediumLayerB2*) by interior routers Switch0, Switch1 and Switch2;

2. Destination *DestinationB* yields *HB_CommonLayer* as destination IPCP. From perspective of HostA this IPCP's name acts as whatevercast name because recursive (N)-to-(N-1) lookup yields three available IPCPs (*hb_MediumLayerB0*, *hb_MediumLayerB1* and *hb_MediumLayerB2*);
3. In order to successfully form flow between *SourceA* and *DestinationB*, three underlying flows between *HA_CommonLayer* and *HB_CommonLayer* going through *S0_CommonLayer*, *S1_CommonLayer* and *S2_CommonLayer* are allocated first (which leads to recursive allocation on *MediumLayerXX* DIFs).
4. Data from *SourceA* to *DestinationB* are passed to *CommonLayer* IPCPs, where are replicated onto three resiliency flows going through *MediumLayerXX* IPCPs.

8.4. Load balancing

8.4.1. Introduction

In order to discuss how load balancing is achieved in RINA it is important to define more accurate terms in which to describe the strategies necessary to support "load balancing" with in a data centre. It is also worth noting that load balancing is architecturally different from load distribution. In *load balancing*, the intermediate node or the load balancer has to keep track of the states of all flows which are being forwarded from this node. The intermediate node which is performing the load balancing could be the single point of failure and a potential bottleneck.

Load distribution is the redirection of flows towards different paths and instances of the same server application. This has the advantage of there being no need for dedicated hardware nor a need to maintain the states of the flows at intermediate nodes. In general, load distribution is achieved by either the distribution of flows across multiple paths towards the same destination and/or distribution of flows towards multiple instances of the same application running on multiple server machines. The load can be distributed on the basis of geographical location of the (client and server) applications or many other parameters such as; the current load on a server, or data and flow requirements of the client application.

In spite of architectural differences, the goal of both, load balancing and load distribution is to ensure service availability and to minimize the response time. The following points outline some example mechanisms for load distribution and load balancing:

Load Distribution

- Creating multiple end to end flows and splitting data packets across these flows from source to destination at transport layer (e.g. MPTCP)
- Routing data packets across multiple paths at network layer (e.g. ECMP routing)
- Selection of geographically nearest server (e.g. Ubuntu mirrors)
- Selection of server with the highest available bandwidth (e.g. netselect function in Ubuntu)
- Re-direct flows to different servers (e.g. DNS)
- In-flow load levelling (TCP congestion control, DCCP). It is neither load distribution nor load balancing. It is a cooperative process to maintain the load on a single path to avoid congestion thus improving the overall network performance and minimizing the response time.

Load balancing

- Deploy an intermediate node at the entry point (POP) of the data centre that can distribute traffic evenly (e.g. load balancing using reverse proxy)
- Load balancing using top of the rack switch (only balance the load among the servers within single rack)
- Deploying new VMs, shutdown selected machines and/or move a VM from one physical machine to another (VM mobility)

8.4.2. Load Balancing in RINA

RINA provides "policy" hooks (which are standardised) and has implicit support for multi-homing, security and QoS. Ideally, load balancing in RINA based data centres should not need to have special hardware or dedicated nodes, and as such in the presence of an efficient load distribution mechanism, the deployment of dedicated load balancers is not necessary. Thus minimizing the installation and operational cost of the data centres.

Thus the load balancing mechanism for RINA based data centres has to be different. Both categories of load balancing can be achieved in RINA. The following diagram outlines how load balancing could work in RINA:

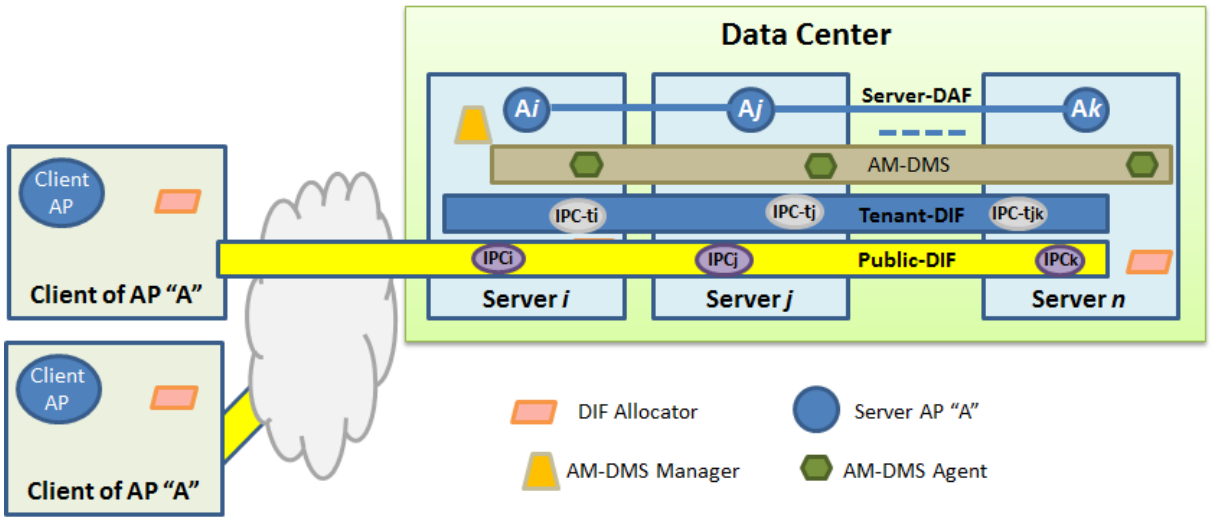

Figure 48. Load Distribution in RINA

With reference to Figure 48, the following statements are made:

1. The application service, i.e. the processes that make up a functional application are a single DAF (called "Server-DAF")
2. The DAF has *N* number of instances, each instance referred to as a Distributed Application Process (DAP).
   - Communication between DAPs in the "Server-DAF" be on a private/ restricted DIF called the "Tenant-DIF".
   - The DAPs share their load statistics with each other (in the "Tenant-DIF") and decide for their own availability to accept further flow requests.
   - Each DAP will also be on a "Public-DIF" through which client APs can get connected with the server APs.
   - Each DAP registers or deregisters with the IPC process in the Public-DIF depending on it's availability to accept new flow requests.
   - All the DAPs are managed by the Application-Management DMS (AM-DMS). (If elastic provisioning is required)

At this point, each DAP can share "load" information, and new DAP's can be started or old ones stopped as necessary (supporting elastic provisioning). How "load" is defined depends on the application specific workload. There

is **no single best** solution for all types of (application process) workloads and flow characteristics and there are multiple ways of achieving it. Each comes with a set of pros and cons and suitable for some special use case scenarios. In this research, the most generic mechanism for load distribution is presented and will eventually result in balancing the load across the multiple paths as well as among the multiple instances of the server application process.

We outline five methods for achieving load balancing. The first two involve the client AP resolving a whatever-cast name (containing the available server DAPs), using a random resolution strategy (see Section 8.4.3) or a "nearest" resolution strategy (see Section 8.4.4).

The second three involve some exchange of load information between server DAPs. These options are discussed in Section 8.4.5, Section 8.4.6 and Section 8.4.7 respectively. A final option is to combine client based methods, with load sharing ones, so a hybrid set of strategies can be used.

All of the methods make use of *Whatever-cast* names. A thorough definition of name-space management is given in Section 8.2. For our purposes a whatever-cast name represents a group of addresses, and has a **single** defined name resolution function.

8.4.3. Option 1: Pseudo Random Selection

Each DAP of the "Server-DAF" advertises itself in whatever-cast name visible on the "Public-DIF". $W_{serverdaf}$ name refers to the whatever-cast name used in the server DAF. It can be defined as follows:

```
class ServerDAFWhatevercast < Whatevercast {
  # Defined resolution function
  begin Address resolve()
     int random = random() * addresses.size()
     # return a single address
     return addresses.get(random)
  end
}
```

So a single address is chosen at random from the available addresses (each corresponding to a server DAP) within the whatever cast name. The client uses a single whatever cast name as the destination for flow requests. The

lookup in the Directory Forwarding Table (DFT), on client AP side, results in the selection of (random) address from the ServerDAFWhatevercast name. The flow allocation request is sent to the IPC-Process (where the server DAP registered to). This IPC process sends the request to DAP (for approval/refusal) and the response is returned to the client AP.

Advs

1. The Public-DIF has to know only the names/addresses of the available server DAP instances.
2. No load information needs to be shared with the clients.
3. Server DAPs can indicate available by registering with the whatever cast name.

Cons

1. The limitation of this approach is that the IPC processes in the Public-DIF do not know about the most current state of the server instance.

Each server application instance has a limited service rate. The IPC processes within the Public-DIF are not aware of this limit. It may happen that IPC processes forward flow allocation requests to a specific server AP instance at a rate larger than its service rate, causing potential overrun. So until the Server-DAF updates the whatever-cast name, and this list of available server instances is propagated, one or more instances could become flooded.

8.4.4. Option 2: Minimum Hop Selection (Geographical Load Distribution)

Each DAP of the "Server-DAF" advertises itself in whatever-cast name $W_{serverdaf}$ visible on the "Public-DIF". The DFT on the client side is not only updated about the instances of the AP which are currently accepting the new flows but also know about the number of physical hops to where these instances are located. In this way the flow requests coming from the client APs could be forwarded to the server AP instance which is located at the minimum number of hops distance.

```
class ServerDAFWhatevercast < Whatevercast {
  # Defined resolution function
  begin Address resolve()
     int     min = MAX_HOPS
     Address a   = addresses[0]
     for each addr in addresses do
       if (addr.hop_count < min) then
         a = addr
         min = addr.hop_count
       end if
     end
     return addr
  end
}
```

Advs

1. The Public-DIF has to know only the names/addresses of the available server DAP instances.
2. Server DAPs can indicate availability by registering with the whatever-cast name.

Cons

1. Hop count information needs to be shared with the clients.
2. Client APs are aware of the number of available server DAPs.
3. Its dependant on the location of clients, thus a cluster of clients on a given network segment all route to the same server DAP (because it is the closest)

The "locality" issue can somewhat mitigated if the server instance is be capable to forward/redirect the flow requests to other server AP instances, see next section.

8.4.5. Option 3: Delegation by a proxy DAP

One method of avoiding sharing "load" information with the client APs directly is to allow client flow requests be routed to a "proxy" DAP instance, which estimates the expected load on the other server DAPs and selects the most appropriate DAP (or N DAPs) to perform the actual work.

DAF load determination

```
# Defined resolution function
begin Address resolve()
   int     min = MAX_LOAD
   RIB.DAP d   = serverdaf[0]
   for each dap in serverdaf do
     if (dap.load < min) then
       d = dap
       min = dap.load
     end if
   end
   return d
end
```

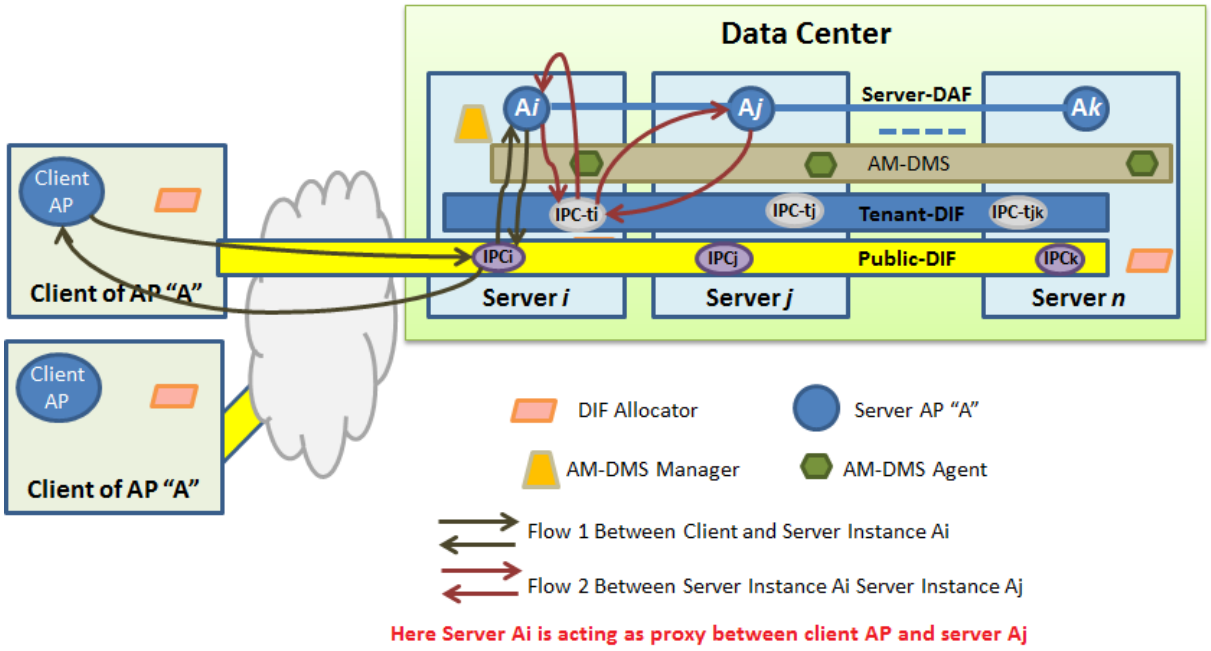

Figure 49. Server DAP Ai acting as proxy node between client and the available server DAP Aj

The server Ai (DAP_i) accepts the incoming flow request. DAP_i consults its internal RIB representation of the "load" on other DAPs, and determines DAP_j has capacity to service the "client" request. It accepts the flow from the client, and either; i) reuses an existing flow from DAP_i to DAP_j or ii) opens a new flow between DAP_i to DAP_j. When a work request arrives from the client it is delegated to DAP_j for processing.

Because these flows are created on the "Tenant-DIF", they are not visible to the client, therefore the number of instances operating in the Server-DAF is hidden.

Advs

1. The Public-DIF has to know only the names/addresses of the available **proxy** DAP instances.
2. Server DAPs can exchange availability and load information.
3. It uses a trusted DIF to exchange load information, where the client AP cannot intercept, or otherwise create mischief.
4. Its not dependant on the location of the client APs.
5. It allows the service provider to "tune" the ratio of proxy DAPs and worker DAPs.

Cons

1. The proxy DAP could itself become the limiting factor, in the number of clients it can support. (limited by available bandwidth or processing speed of the proxy)
2. The accuracy of the "load" information is critical to the success of this mechanism.

In general, this solution has a lot of advantages and has most of the desirable criteria. However, this is the most efficient where the ratio of proxy DAPs and work DAPs is large. If the QoS requirements on the flows are demanding, it is possible to run out of resources at the proxy DAP, while there is plenty of capacity in the "worker" DAPs.

In essence, it means two server DAP's are involved in processing every client request. Indeed in some scenarios, video streaming for example, the proxy is merely accepting packets from the "streaming worker" DAP, and forwarding them over another flow to the client AP. Although practical, it may not be the most efficient and cost effective way of providing these services.

8.4.6. Option 4: Redirection using Client AP

Using this method the "load" is calculated in a similar method to the algorithm in DAF load determination.

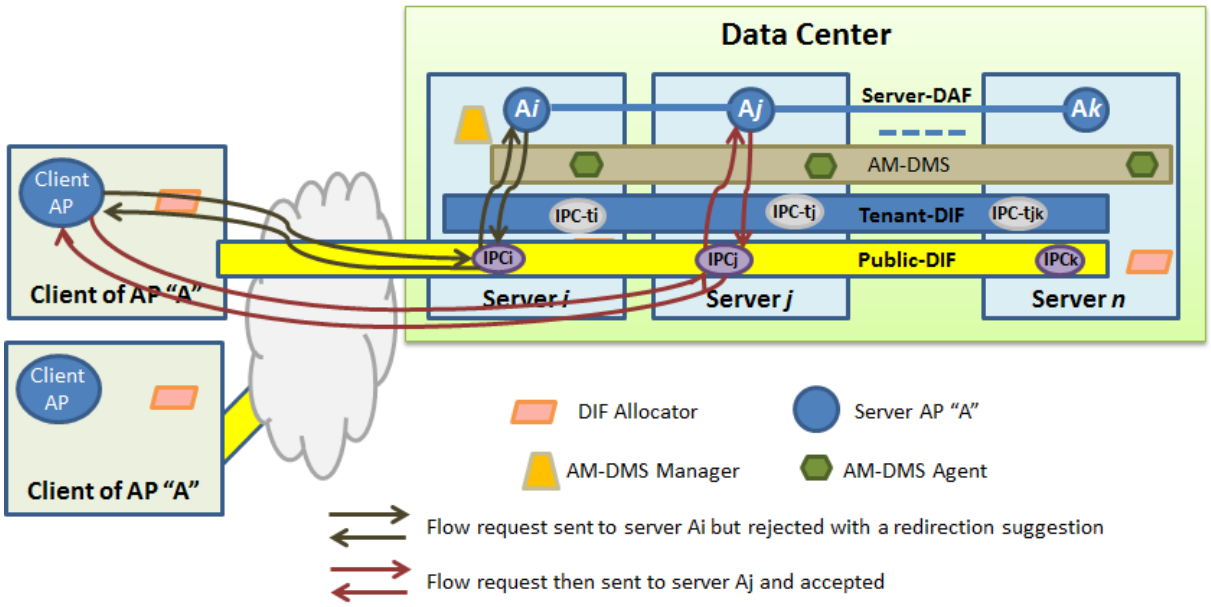

Figure 50. Server Ai rejects with a new suggestion which is told to the client.

The client will be told by the server DAP_i about the address of the next available instance so that the client itself initiates a new flow request with the other server DAP_j. Client then connects with the available server DAP_j. This method could potentially expose some information about the current load on the server instances.

Advs

1. The Public-DIF has to know only the names/addresses of **some of** available DAP instances.
2. Server DAPs can exchange availability and load information.
3. It uses a trusted DIF to exchange load information, where the client AP cannot intercept, or otherwise create mischief.
4. Its not dependant on the location of the client APs.

Cons

1. It requires that the flow allocation response can contain a "hint" of a more suitable DAP.
2. The accuracy of the "load" information is critical to the success of this mechanism.
3. The existence of DAPs is eventually, passed to the client AP's using them.

There are two methods to providing a hint. One is to refuse the connection, but explicitly send a "neighbour" advertisement to the client, announcing

the availability of DAP_j. This could lead to a data-race, where the neighbour advertisement arrives after the result of the flow allocation. Thus the preferred suggestion is to encode an alternate address in the flow allocation response, so as the "hint" is available when the flow allocation response is processed at the client AP. If the client AP used a whatever-cast name to find DAP_i and DAP_j is also a member of the whatever-cast name, then the client can retry the flow allocation request (if the DAF policy permits it) on the address of DAP_j.

8.4.7. Redirection using a DAF policy

Using this method the "load" is calculated in a similar method to the algorithm in DAF load determination.

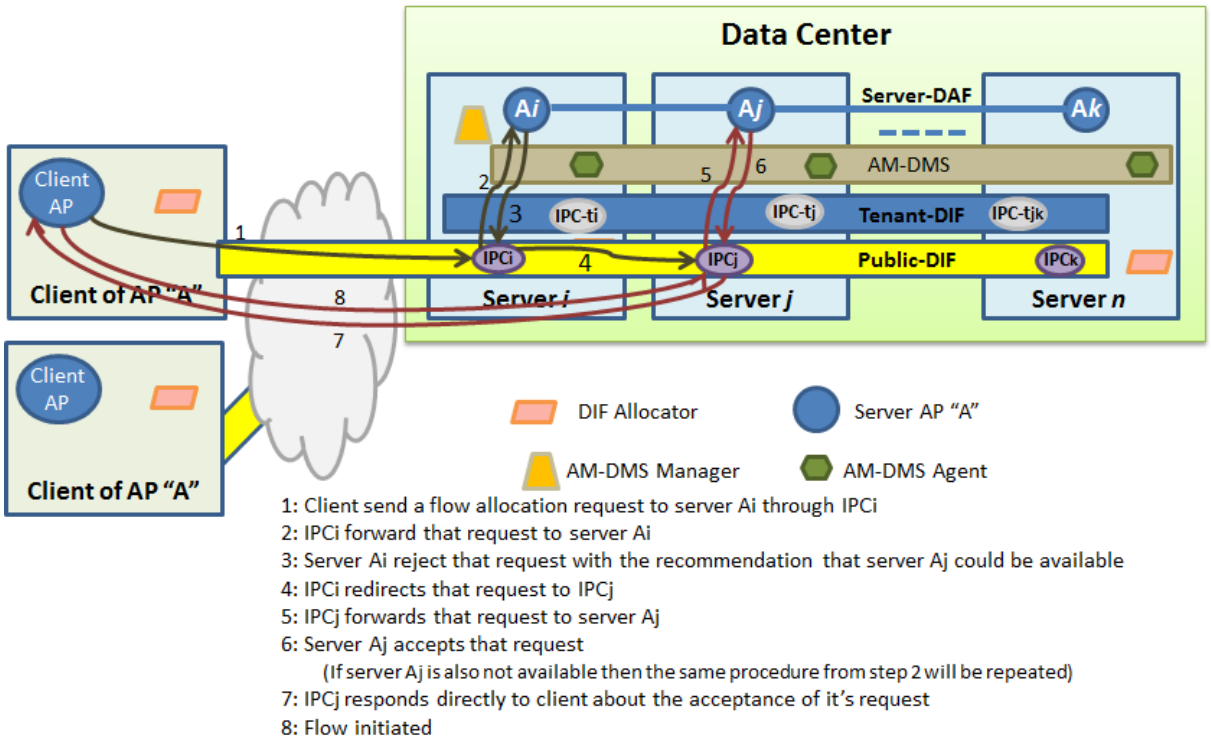

Figure 51. Server Ai rejects with a new suggestion.

The IPC_i then forwards the same request to DAP_j via IPC_j, which then responds directly to the client with the result. However it would require some extensions in Flow Allocator and IPC process implementation. In essence, we are rewriting the flow allocation request with a different destination address (DAP_j). Thus is a possibility of a rogue DAP redirecting flow requests as a form of DOS attack, is small as the rogue DAP would have authenticate to the DIF and comply with DIF policies.

Advs

1. The Public-DIF has to know only the names/addresses of **some of** available DAP instances.
2. Server DAPs can exchange availability and load information.
3. It uses a trusted DIF to exchange load information, where the client AP cannot intercept, or otherwise create mischief.
4. Its not dependant on the location of the client APs.
5. Transparent operation for the client APs.

Cons

1. It requires that the flow allocation response can contain a "hint" of a more suitable DAP.
2. The accuracy of the "load" information is critical to the success of this mechanism.
3. Potentially, DAP_j may decide to forward the request to DAP_k, so a hard limit needs to be placed on the number of forwards.
4. More complex to implement.

From the client AP perspective, this is transparent in operation. However, from the DAF/IPCP viewpoint this is the most complex to implement.

8.4.8. Conclusion and Future Work

Load distribution in RINA not only could distribute the work load of the server instances but also capable of evenly balancing the network load. Poorly selected server instances could cause adverse effects on the network load and congestion could occur in the network. Therefore the selection of server instances from the available list must be as smart as possible. There are countless selection methods available in order to select one server instance out of a list. In this text, only two methods, random selection and min-hop selection are presented.

The plan is to compare some other methods as well e.g. QoS requirements based server instance selection, server machine capabilities based selection etc. A modified load distribution algorithm will also be used in future

experiments in which the algorithm can do one of the following two options:

- Select the nearest server instance just like min-hop method and if that instance is not available at the moment then the instance will forward/ redirect the request to the next nearest instance.
- Make one server instance as root and forward the flow request to the root. Everyone in the public network knows only the address of the root. The root will forward/redirect to it's leaves hierarchically until the flow request found a more suitable instance.

8.5. Providing Parallel flows over the shim DIFs

The shim DIF over Ethernet 802.1Q developed in the IRATI project has the limitation that it can support only a single connection between two Ethernet endpoints (i.e. MAC addresses), and thus it can register only a single name and support only a single flow between a pair of port_ids. [Vrijders2013]. This limitation was accepted by IRATI because the goal was to support only IPC processes on top of this shim DIF.

The PRISTINE project advanced the management aspects of RINA and found the need to distinguish between management flows and data transfer flows. [7] So a normal IPCP needs to establish two different flows, which is unsupported by the current shim DIF.

This section specifies the final specification from PRISTINE for two shim DIFs. The first is a shim DIF over Ethernet with LLC, which can use Service Access Points (SAPs) to distinguish connection endpoints, and thus provide parallel connections and ultimately parallel flows between Application Process Instances [8]. The second is an updated shim DIF over UDP, that uses the same method of operations, with UDP ports instead of SAPs. The second shim DIF was partially implemented in user-space to validate that the method of operation in these two specifications is sound. A full shim DIF over LLC for IRATI may be implemented as part of the ARCFIRE project.

[7] https://github.com/IRATI/stack/issues/748
[8] https://github.com/IRATI/stack/issues/579

8.5.1. Specification for the Shim DIF over Ethernet with Link Layer Control (LLC)

Introduction

This section specifies the final specification from PRISTINE for a shim DIF over LLC (IEEE 802.2 [8022]) which in turn uses the Ethernet (IEEE 802.3 [8023]) layer. It uses Type I LLC with unnumbered (U-format) frames. Given that a RINA DIF expects a RINA API as the lower API, the purpose of a Shim DIF is to create as thin a veneer as possible over a legacy protocol to allow a RINA DIF to use it without modification. The goal is not to make legacy protocols provide full support for RINA and so the shim DIF should provide no more service or capability than the legacy protocol provides.

An Ethernet shim DIF spans a single Ethernet "segment". This means relaying is done only on 0-DIF addresses and that the scope of the shim DIF is small.

Type I LLC + Ethernet comes with the following limitations, which are reflected by the capabilities provided by the shim DIF:

- There is no explicit flow allocation. We define a set of Ethernet frames as belonging to the same flow if the source/destination MAC addresses and the Destination Service Access Point (DSAP) and Source SAP (SSAP) are the same.
- Ethernet doesn't perform fragmentation and reassembly functions.
- There are no guarantees on reliability (Type I LLC).
- The maximum supported payload size is 1500 bytes. LLC does not support jumbo frames.
- There is no minimum payload size.

It is recommended that the user of the shim DIF is a normal IPC process. Other applications could also make use of the shim DIF but could require modification to comply with these restrictions.

Use of the LLC and Ethernet Header

IEEE 802.3 Ethernet frame with 802.2 payload

Table 10. IEEE 802.3 with IEEE 802.2

Preamble (7 bytes)	Start of frame delimiter	Destination MAC address 96 bytes)	Source MAC address (6 bytes)	Length (2 bytes)
IEEE 802.2 LLC Header (3-4 bytes)	Payload (M ⇐ 1500 bytes)	PAD	Frame Check Sequence (4 bytes)	Interframe gap (12 bytes)

- Destination MAC address: The MAC address assigned to the Ethernet interface the destination shim IPC Process is bound to.
- Source MAC address: The MAC address assigned to the Ethernet interface the source shim IPC Process is bound to.
- Length: Sum of the length of the LLC header and the payload (i.e excluding Preamble, SD, FCS)
- Payload: This Shim IPCP has a maximum supported payload of 1500 bytes, providing fragmentation when needed is the responsibility of the upper layer DIF. Oversized packets will be dropped.
- PAD: if the payload is less than 46 bytes, this field adds padding to reach a minimum frame length of 64 bytes.
- Frame Check Sequence: a 32-bit CRC calculated of the entire frame.

IEEE 802.2 LLC header

Table 11. IEEE 802.2 header

DSAP address (1 byte)	SSAP address (1 byte)	Control (1 or 2 bytes)

SAPs will be used to distinguish between different connections. These SAP assignments only have to be unique within their scope and are not required to be globally unique. A SAP has the same function as a CEP-id. During flow allocation they are mapped to a port-id. A port-id is the identifier of a flow and forms the boundary between the Shim DIF and a 1-DIF.

Table 12. SAP assignments

Least significant bit (DSAP)	Least significant bit (SSAP)

I/G[a]	X[b]	Data (6-bits)	C/R[c]	X[d]	Data (6 bits)

[a] Individual / Group Bit
[b] Bit reserved by ISO definition
[c] Command Response Bit
[d] Bit reserved by ISO definition

As can be seen in the table above, not all SAPs are available for use. The least significant bit of the DSAP identifies whether it is a group or individual SAP address. We will only use individual SAPs. For the SSAP it identifies whether it is a Command or Response [8022]. We will not use this bit since we are not using Data State Vectors. The second least significant bit will also be left 0, due to the ISO definition. All the other bits are free to be chosen which means that in total 64 SAP addresses can be used within the Shim DIF between 2 application processes (APs). For simplicity, we will set this range as 0x00 - 0x40. The NULL and global SAPs will not be used. This shim has a special SAP 13 (0x0D) which is reserved for listening for new incoming flows.

Directory function

The shim DIF over LLC uses ARP, which is the L2 protocol that performs this function in Ethernet.

Use of the Address Resolution Protocol (ARP)

The shim DIF over LLC uses ARP in request/response mode to resolve a network layer address into a link layer address instantiating the state for a flow to this protocol. In effect, in the context of the shim DIF, it must map a registered name in the DIF to a shim IPC Process address (MAC Address) and instantiating a MAC protocol machine equivalent of DTP. ARP provides the mapping of the (N)-address of the node to the (N-1)-address of the point of attachment. The relation between these different protocol machines can be seen in the following figure:

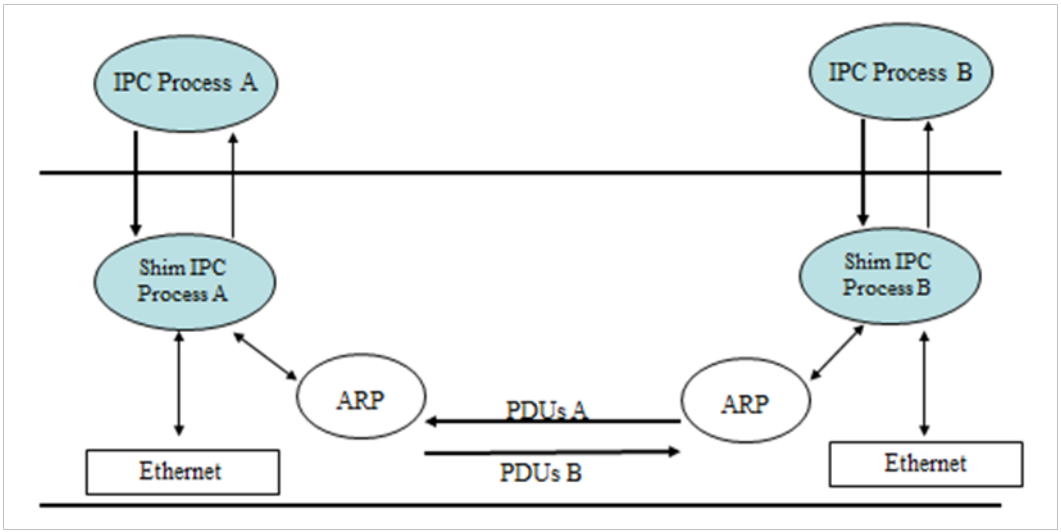

Figure 52. Relation between protocol machines

Although the ARP specification provides the capability to map network layer addresses of any length to link layer addresses of any length, ARP implementations usually do not conform to the standard and are limited to map IPv4 addresses to MAC addresses. This specification assumes an ARP implementation compliant with RFC 826 [RFC826].

The ARP message has one field for the length of the network protocol address which is one byte long. This means a network protocol address, thus any name registered with the shim DIF, can't be longer than 255 characters (assuming ASCII).

The fact that there is only one field to signify the length of the network protocol address also entails that names registered with the DIF have to be equally long on a per packet basis, since there is only one field to specify the length of both "network protocol addresses". To solve this, the length field is filled in with the size of the longest name in the packet. The shorter name is grown with padding to the size of the longest name. The padding is removed on the receiving side. The padding is defined as the zero byte (0x00). Padding causes unnecessary packet size overhead, but since ARP packets are only sent the first time a flow allocation is done to a destination name, and never in the actual data transfer phase, this overhead is negligible.

Service Definition

This DIF offers only one QoS cube. It is defined as follows:

Table 13. QoS cube of the shim DIF

ID	1
Name	Unreliable
Average bandwidth	Depends on standard
Average SDU bandwidth	Depends on standard
Peak bandwidth-duration	Depends on standard
Peak SDU bandwidth-duration	Depends on standard
Burst period	Depends on standard
Burst duration	Depends on standard
Undetected bit error rate	Depends on standard
Partial delivery	Allowed
Order	No
Max allowable gap in SDUs	Any
Delay	Depends on standard
Jitter	Depends on standard

Configuration

Each shim IPC Process is assigned to an Ethernet interface. The shim IPC Process listens on a reserved SAP for new flow allocation requests, defined as SAP 13 (0x0D). This shim IPC process allows multiplexing legacy 802.3 and Ethernet II traffic on the assigned Ethernet interface, but we recommend using the shim DIF over Ethernet with LLC over a VLAN interface to easily separate RINA traffic.

There is a certain amount of information that the shim IPC Process needs in order to operate effectively. This information includes:

- Shim IPC Process info: This includes the shim IPC process AP name and instance ID, the OS specific name of the Ethernet interface to be bound to, and a shim DIF name. Note that the shim DIF name is only of local significance in the processing system and used to discern between different Ethernet segments available to this system. All shim IPCPs that are on a specific 802.3 segment are in the same shim DIF, regardless the shim DIF name given to them.
- Directory: The directory provides the shim IPC process with information about which name can be accessed from each of the shim IPC processes in the same shim DIF. The directory consists of entries mapping <name> to <MAC address>. In the shim DIF for Ethernet the

Address Resolution Protocol – ARP – is used to perform the directory function.

Bootstrapping

When the shim IPC Process is created, it makes the necessary arrangements with the OS in order to receive the Ethernet traffic of the shim DIF directed to the MAC address of the Ethernet interface it is bound to. It will listen on SAP 0x0D (13) for new flow allocation requests.

Enrollment

All shim IPC processes in the same Ethernet segment are assumed to be part of the same shim DIF. There is no explicit enrollment procedure.

Application registration/unregistration

When a name is registered with the shim IPC process, the operation is accepted or denied depending on a set of rules for access control. When the name is registered, the Shim IPC Process' ARP cache gets populated with an entry, mapping the <name> to the <MAC address> of the interface the shim IPC Process is bound to. When a name is unregistered, the shim IPC process removes the corresponding entry from the ARP cache, so future queries of the name are ignored.

The Shim IPCP assumes the following API towards the Ethernet ARP table:

- arpAdd(netaddr).submit: Adds a mapping of a registered name to the MAC address of the interface to the ARP table. It returns 'success' if the mapping was added, 'failure' if it was not.
- arpRemove(netaddr).submit: Removes a mapping of a registered name to the MAC address of the interface in the ARP table of the interface. If it is removed 'success' is returned, else 'failure' is returned.

When using the ARP protocol to resolve destination names to MAC addresses:

- arpMapping(netaddr).submit : Requests ARP for a mapping of a network address to a MAC address, when the mapping is found, arp_mapping(netaddr, hwaddr).deliver is called.

port_id states definition:

- NULL - This state indicates that the port_id cannot be used.
- SND_PENDING - This state indicates that the flow is waiting for an ARP reply to arrive.
- DST_PENDING - This state indicates that the flow is waiting for an allocate_response from a user of the shim IPCP.
- SAP_PENDING - This state indicates that the flow is waiting for the arrival of an Ethernet frame from the destination to fill in the DSAP in the structure holding the flow information.
- ALLOCATED - This state indicates that the flow is allocated and the port_id can be used to read/write data from/to.

applicationRegister(name).submit

When invoked

This primitive is invoked to register a name with the shim DIF. ARP does not differentiate between client/server, which means every name has to be available in the ARP table, even names associated with client applications. A SSAP may be reserved to be used for this registered name. If not, a new SSAP will be chosen for each flow allocation (this reduces the maximum amount of possible flows between two MAC addresses from 4096 to 64).

Action upon receipt

arpAdd(name).submit is called. If successful, a mapping of the name to the hardware address of the device is added in the ARP table of the interface. If it fails, an error is generated.

applicationUnregister(name).submit

When invoked

This primitive is invoked to deregister a name on top of the shim IPC process. This deregisters the name in the shim DIF and removes it from the ARP table.

Action upon receipt

arpRemove(name).submit is called. If successful, the mapping of the name to the hardware address of the device is removed from the ARP table of the device. If it fails, an error is generated.

Flow Allocation

allocateRequest(name).submit

When invoked

This primitive is invoked by a source application instance to request a new flow. It specifies the destination name, and, optionally, the source AP name and the source AE name.

Action upon receipt

A shim over LLC may or may not support parallel flows between the same pair of application processes. If the request is accepted, a new flow is allocated, a port-id is created in the NULL state and arpMapping(netaddr).submit is called. The port_id transitions to the SND_PENDING state. If the request is rejected, a negative allocateResponse(reason).deliver is returned.

arpMapping(netaddr, hwaddr).deliver

When invoked

This is invoked by the ARP protocol machine when a requested mapping has been added to the ARP table. The Shim IPC process is supplied with the mapping of the registered name to an Ethernet hardware address (hwaddr).

Action upon receipt

If the port_id is in the SND_PENDING state, (there is an outstanding allocateRequest(name).submit), an allocateResponse(reason).deliver is invoked, and the Ethernet hardware address (MAC address) is stored for this flow. A new free SAP for the combination of the source and destination MAC addresses is chosen and added to the structure holding information related to this flow. The port_id transitions to the SAP_PENDING state. If the port_id is in another state, nothing happens.

Ethernet frame

When invoked

When the port-id is in the SAP_PENDING state, an Ethernet Frame containing the source AP name may be sent from the chosen SSAP to the reserved SAP (0x0D) for flow allocation. When the port_id is ALLOCATED, Ethernet frames containing SDUs may be sent. Otherwise, SDUs are dropped.

Action upon receipt

When there is no flow for the source and destination MAC address and SSAP, or parallel flows are allowed, and the DSAP is 0x0D , a new port_id is created. The hardware addresses and SSAP for this flow are stored and the port_id transitions to DST_PENDING. allocateRequest(name).deliver is called. If there is no flow for the source and destination MAC address and SSAP, and DSAP is not 0x0D, the frame is dropped.

If there is a flow for the source and destination MAC address and the DSAP of the frame and if the port-id is in SAP_PENDING, then the SSAP of the frame is used to add the DSAP to the structure holding the details of the flow. The port-id transitions to ALLOCATED. write.deliver is called.

If there is a flow for the source and destination MAC address and SSAP and DSAP, and the port_id is in the ALLOCATED state, write.deliver is called. If there is a flow for the source and destination MAC address and SSAP and DSAP, and the port_id is not in the ALLOCATED state, the frame is dropped.

In any other case, the frame is dropped.

allocateResponse(reason).submit

When invoked

This primitive is invoked by the destination application in response to an allocateRequest(name).deliver.

Action upon receipt

If the port-id is not in the DST_PENDING state, an error is generated. If it is in the DST_PENDING state and if the allocate response is positive, a flow has been established for this port_id; any queued frames are delivered to the destination, and a free SAP is chosen for this flow. The port_id transitions to the ALLOCATED state. If the allocate response is negative,

the flow creation procedure rolls back and a variable reason may be returned to indicate the failure reason. The port_id transitions to the NULL state.

Read.submit

When invoked

This is invoked by the application when it wants to read a SDU.

Action upon receipt

When the shim IPC process receives this primitive in the ALLOCATED state, it will wait for the next Ethernet frame to arrive, or deliver any outstanding SDUs. It is assumed neither fragmentation nor concatenation are performed by the shim IPC Process; therefore each Ethernet frame transports one and only one SDU. In any other state this operation is invalid.

Write.submit

When invoked

This is invoked by the application when it wants to send one or more SDUs.

Action upon receipt

When the shim IPC process receives this primitive and the port_id is in the SAP_PENDING, the source and destination MAC address are filled in in the Ethernet frame, and the SSAP and DSAP are set to 0x0D and the frame is passed to the OS for delivery. In the ALLOCATED state, it will create an Ethernet frame with source and destination MAC addresses and the correct source and destination SAPs, and pass it to the OS for delivery. It is assumed neither fragmentation nor concatenation are performed by the shim IPC Process, therefore the shim IPC Process uses a different Ethernet frame for each SDU passed to it. It is the responsibility of the upper layer DIF to provide SDUs of adequate size in order to allow for an efficient use of the Ethernet medium. In any other state, the frame is discarded and an error is generated.

Deallocate.submit

When invoked

This service primitive is invoked by the application to discard all state regarding this flow. It is the responsibility of both the source and destination application to invoke this primitive. It is a local event.

Action upon receipt

When the shim IPC process receives this primitive, the port_id transitions to the NULL state.

State diagram

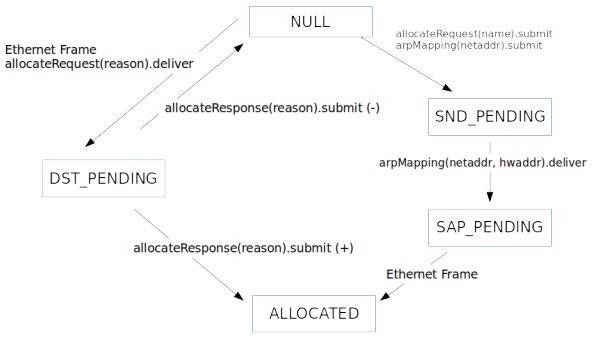

Figure 53. State diagram of the shim DIF

8.5.2. Shim DIF over UDP with DNS

We inform the reader that this specification is highly similar to the Shim LLC specification above. We chose to implement parts of this shim DIF to validate both the UDP and LLC shim DIF, as the implementation effort needed to validate the functionality is much lower.

8.5.3. Specification for the Shim DIF over IPv4/UDP with Domain Name System (DNS) Support

This section specifies the final specification from PRISTINE for a shim DIF over UDP which in turn uses Domain Name System to update the registry. Given that a RINA DIF expects a RINA API as the lower API, the purpose of a Shim DIF is to create as thin a veneer as possible over a legacy protocol to allow a RINA DIF to use it without modification. The goal is not to make legacy protocols provide full support for RINA and so the shim DIF should provide no more service or capability than the legacy protocol provides.

A UDP shim DIF spans the whole Internet. This means relaying is done on IP addresses and that the scope of this shim DIF is considered global.

UDP comes with the following limitations, which are reflected by the capabilities provided by the shim DIF:

- There is no explicit flow allocation. We define a set of datagrams as belonging to the same flow if the source/destination IP addresses and the source and destination UDP ports are the same.
- UDP does not provide error correction (although the L2 supporting UDP may), sequencing, duplicate elimination, flow control, or congestion control.
- There are no guarantees on reliability.
- The maximum supported payload is 65,507 bytes (65,535 as imposed by the 16 bit Length field – 8 byte UDP header – 20 byte IP header). However, we recommend to stay within the limits of the L2 MTU on LANs and below 508 bytes when using the public internet for reliability.
- The minimum payload size depends on the L2 technology. For Ethernet II, the minimum payload is 46 bytes, UDP payloads smaller than 16 bytes need to be padded.

It is recommended that the user of the shim DIF is a normal IPC process. Other applications could also make use of the shim DIF but could require modification to comply with these restrictions.

Use of the IPv4 and UDP Header

IPv4 datagram

Table 14. IPv4 datagram

VERS	LEN	TOS	Total Length	
Identification		Flags	Fragment Offset	
TTL	Protocol number	Header Checksum		
Source IP Address				
Destination IP Address				
Options				
Padding				

- VERS: IP version, the shim DIF uses IPv4 (4). 4 bits.
- LEN: The length of the IP header in 4-octet quantities. 4 bits.
- ToS: Type of Service. 8 bits.
- Total length: headers + data. 16 bits
- Identification: ID for fragmentation and reassembly. 16 bits.
- Flags: control flags. 3 bits: 1) reserved. 2) DF: don't fragment. 3) MF: More Fragments.
- Fragment offset: needed for reassembly. 13 bits.
- TTL: time to live. Decreased by 1 at each hop. 8 bits.
- Protocol: The shim DIF uses UDP (17). 8 bits.
- Header Checksum. Checking integrity of the IP header. 16 bits.
- Source IP address: The IP address assigned to the interface that the source shim IPC Process is bound to. 32 bits.
- Destination IP address: The MAC address assigned to the interface the destination shim IPC Process is bound to. 32 bits.
- Options: various options, not used by the shim DIF.
- Padding: This field adds padding to reach a minimum frame length for L2 if needed.

UDP header

Table 15. UDP header

Source Port	Destination Port
Length	Checksum
	DATA

- Source Port: The shim DIF uses ports in the ephemeral range (depends on operating system). 16 bits.
- Destination Port: The shim DIF uses uses ports in the ephemeral range (depends on operating system). 16 bits.
- Length: UDP header + DATA: 16 bits.

UDP ports will be used to distinguish between different connections. These UDP port assignments only have to be unique within their scope (a host as identified by an IP address) and are not required to be globally unique. A UDP port has the same function as a CEP-id. During flow allocation they are mapped to a port-id. A port-id is the identifier of a flow and forms the boundary between the Shim DIF and a 1-DIF.

Directory function

The shim DIF over UDP uses DNS, which is the infrastructure that performs this function in IP networks.

Use of the Domain Name System (DNS)

The shim DIF over UDP uses Dynamic DNS (DDNS) to register and resolve domain names to IP addresses instantiating the state for a flow to this protocol. In effect, in the context of the shim DIF, it must map a registered name in the DIF to a shim IPC Process address (IP Address) and instantiating a IP protocol machine equivalent of DTP. DNS provides the mapping of the (N)-address of the node to the (N-1)-address of the point of attachment.

The shim DIF for UDP takes a conservative approach to naming, adopting the naming conventions of RFC 1035:

- A host name (label) can start or end with a letter or a number.
- A host name (label) MUST NOT start or end with a '-' (dash).
- A host name (label) MUST NOT consist of all numeric values.
- A host name (label) can be up to 63 characters.
- The total name can be up to 255 characters.

Service Definition

This DIF offers only one QoS cube. It is defined as follows:

Table 16. QoS cube of the shim DIF

ID	1
Name	Unreliable

Average bandwidth	unspecified
Average SDU bandwidth	unspecified
Peak bandwidth-duration	unspecified
Peak SDU bandwidth-duration	unspecified
Burst period	unspecified
Burst duration	unspecified
Undetected bit error rate	unspecified
Partial delivery	Allowed
Order	No
Max allowable gap in SDUs	Any
Delay	unspecified
Jitter	unspecified

Configuration

Each shim IPC Process is assigned to an IP address. The shim IPC Process listens on a reserved UDP port for new flow allocation requests, defined as 3359 (0x0D1F)[9]. This shim IPC process allows multiplexing legacy UDP and IP traffic on the assigned IP address.

There is a certain amount of information that the shim IPC Process needs in order to operate effectively. This information includes:

- Shim IPC Process info: This includes the shim IPC process AP name and instance ID, the IP address to be bound to, the IP address of the DDNS server to send registration info to, and a shim DIF name. Note that the shim DIF name is only of local significance in the processing system and used to discern between different IP addresses available to this system. All shim IPCPs that are able to reach each other are in the same shim DIF, regardless the shim DIF name given to them. This means that, in essence, there is only one UDP shim DIF. However, if the IP address is on an IP subnet without gateways, multiple UDP shim DIFs can coexist.
- Directory: The directory provides the shim IPC process with information about which name can be accessed from each of the shim IPC processes in the same shim DIF. The directory consists of entries mapping <name> to <IP address>. In the shim DIF for UDP with DNS,

[9] Although this is a reserved port (wg-netforce), we think it is fitting and ignore the IANA assignment.

the Dynamic Domain Name System is used to perform the directory function.

Bootstrapping

When the shim IPC Process is created, it makes the necessary arrangements with the OS in order to receive the IP traffic of the shim DIF directed to the IP address of the Ethernet interface it is bound to. It will listen on UDP port 0x0D1F (3359) for new flow allocation requests.

Enrollment

All shim IPC processes reachable from the bound IP addresses are assumed to be part of the same shim DIF. There is no explicit enrollment procedure.

Application registration/unregistration

When a name is registered with the shim IPC process, the operation is accepted or denied depending on a set of rules for access control. When the name is registered, the Shim IPC Process' will register the name in a dedicated DDNS server for this DIF, mapping the <name> to the <IP address> of the interface the shim IPC Process is bound to. When a name is unregistered, the shim IPC process contacts the DNS server to remove the corresponding entry from the DNS system, so future queries of the name are ignored.

The Shim IPCP assumes the following API towards the DNS system:

- dnsAdd(name).submit: Adds a mapping of a registered name to the DNS server with the IP address of the host. It returns 'success' if the mapping was added, 'failure' if it was not.
- dnsRemove(name).submit: Removes a mapping of a registered name to the IP address of the interface in the DNS system. If it is removed 'success' is returned, else 'failure' is returned.
- dnsMapping(name).submit : Requests the DNS system for a mapping of a registered name to an IP address, when the mapping is found, dns_mapping(name, ip_addr).deliver is called.

port_id states definition:

- NULL - This state indicates that the port_id cannot be used.

- SND_PENDING - This state indicates that the flow is waiting for an DNS reply to arrive.
- DST_PENDING - This state indicates that the flow is waiting for an allocate_response from a user of the shim IPCP.
- UDP_PENDING - This state indicates that the flow is waiting for the arrival of a UDP datagram from the destination to fill in the Destination UDP port in the structure holding the flow information.
- ALLOCATED - This state indicates that the flow is allocated and the port_id can be used to read/write data from/to.

applicationRegister(name).submit

When invoked

This primitive is invoked to register a name with the shim DIF. A UDP port may be reserved to be used for this registered name. If not, a new UDP port will be chosen for each flow allocation (this reduces the maximum amount of possible flows between two IP addresses to minimum of the number of ephemeral UDP ports available on the two communicating hosts instead of the product).

Action upon receipt

dnsAdd(name).submit is called. If successful, a mapping of the name to the IP address of the device is added in the DNS system. If it fails, an error is generated.

applicationUnregister(name).submit

When invoked

This primitive is invoked to unregister a name on top of the shim IPC process. This unregisters the name in the shim DIF and removes it from the DNS system.

Action upon receipt

dnsRemove(name).submit is called. If successful, the mapping of the name to the IP address of the device is removed from the DNS table of the device. If it fails, an error is generated.

Flow Allocation

allocateRequest(name).submit

When invoked

This primitive is invoked by a source application instance to request a new flow. It specifies the destination name, and, optionally, the source AP name and the source AE name.

Action upon receipt

A shim over UDP may or may not support parallel flows between the same pair of application processes. If the request is accepted, a new flow is allocated, a port-id is created in the NULL state and arpMapping(netaddr).submit is called. The port_id transitions to the SND_PENDING state. If the request is rejected, a negative allocateResponse(reason).deliver is returned.

dnsMapping(name, ip_addr).deliver

When invoked

This is invoked by the DNS protocol machine when a requested mapping has been received from the DNS system. The Shim IPC process is supplied with the mapping of the registered name to an IP address (ip_addr).

Action upon receipt

If the port_id is in the SND_PENDING state, (there is an outstanding allocateRequest(name).submit), an allocateResponse(reason).deliver is invoked, and the IP address is stored for this flow. A new free UDP port for the combination of the source and destination IP addresses is chosen and added to the structure holding information related to this flow. The port_id transitions to the UDP_PENDING state. If the port_id is in another state, nothing happens.

IP + UDP datagram

When invoked

When the port-id is in the UDP_PENDING state, a datagram containing the source AP name may be sent from the chosen UDP port to the reserved

port 3359 (0x0D1F) for flow allocation. When the port_id is ALLOCATED, datagrams containing SDUs may be sent. Otherwise, SDUs are dropped.

Action upon receipt

When there is no flow for the source and destination IP address and Source UDP port, or parallel flows are allowed, and the Destination UDP port is 3359 (0x0D1F) , a new port_id is created. The IP address and Source UDP port for this flow are stored and the port_id transitions to DST_PENDING. allocateRequest(name).deliver is called. If there is no flow for the source and destination IP address and Source UDP port, and Destination UDP port is not 3359 (0x0D1F), the datagram is dropped.

If there is a flow for the source and destination IP address and the Destination UDP port of the datagram and if the port-id is in UDP_PENDING, then the Source UDP port of the datagram is used to add the Destination UDP port to the structure holding the details of the flow. The port-id transitions to ALLOCATED. write.deliver is called.

If there is a flow for the source and destination IP address and UDP port pair, and the port_id is in the ALLOCATED state, write.deliver is called. If there is a flow for the source and destination IP address and and UDP port pair, and the port_id is not in the ALLOCATED state, the datagram is dropped.

In any other case, the datagram is dropped.

allocateResponse(reason).submit

When invoked

This primitive is invoked by the destination application in response to an allocateRequest(name).deliver.

Action upon receipt

If the port-id is not in the DST_PENDING state, an error is generated. If it is in the DST_PENDING state and if the allocate response is positive, a flow has been established for this port_id; any queued datagrams are delivered to the destination, and an available UDP port is chosen for this flow. The port_id transitions to the ALLOCATED state. If the allocate response is

negative, the flow creation procedure rolls back and a variable reason may be returned to indicate the failure reason. The port_id transitions to the NULL state.

Read.submit

When invoked

This is invoked by the application when it wants to read a SDU.

Action upon receipt

When the shim IPC process receives this primitive in the ALLOCATED state, it will wait for the next datagram to arrive, or deliver any outstanding SDUs. In any other state this operation is invalid.

Write.submit

When invoked

This is invoked by the application when it wants to send one or more SDUs.

Action upon receipt

When the shim IPC process receives this primitive and the port_id is in the SAP_PENDING, the source and destination IP address are filled in in the IP datagram and the source UDP port, the destination UDP port is set to 3359 (0x0D1F) and the datagram is passed to the OS for delivery. In the ALLOCATED state, it will create an IP datagram with source and destination IP addresses and the correct source and destination UDP ports, and pass it to the OS for delivery. In any other state, the datagram is discarded and an error is generated.

Deallocate.submit

When invoked

This service primitive is invoked by the application to discard all state regarding this flow. It is the responsibility of both the source and destination application to invoke this primitive. It is a local event.

Action upon receipt

When the shim IPC process receives this primitive, the port_id transitions to the NULL state.

State diagram

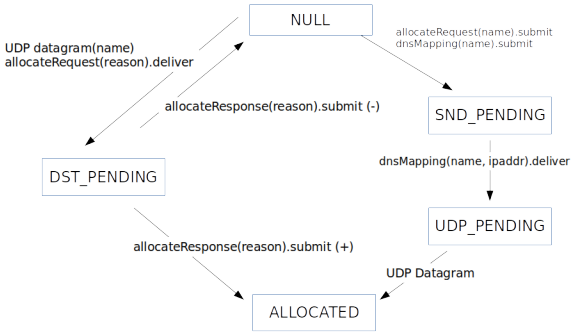

Figure 54. State diagram of the shim DIF

8.5.4. Implementation

The shim IPC process over UDP was partly implemented in userspace to have enough functionality to validate the concepts underpinning its operations. The main reason is that communication with a DNS server from kernel space is a development effort considered outside the scope of PRISTINE. Our proof-of-concept implementation calls the executable binaries `nslookup` and `nsupdate` (packaged together with the BIND DNS server binaries) [10]. We provide a code snippet that shows how communication with the DNS server is performed. The example given is focused on the dynamic update of an entry of the DNS server. The lookup function was implemented in a similar way (a different DDNS command `/bin/nsupdate` is sent to the operation).

```
/*
 * Takes a command and sends it to the DDNS
 * For instance, to update an entry, one would send:
 * "server <DNS name>\nupdate add <AP name> <TTL of the record> A <IP
address>\nsend\nquit\n"
 */
static int ddns_send(char * cmd)
```

[10] https://www.isc.org/downloads/bind/

```
{
        pid_t pid = 0;
        int wstatus;
        int pipe_fd[2];
        char * argv[] = {"/bin/nsupdate", 0};
        char * envp[] = {0};

        /* Create a pipe */
        pipe(pipe_fd);

        /* Fork the process */
        pid = fork();

        /* Override the child with the nsupdate executable */
        if (pid == 0) {
                /* Close one end of the pipe */
                close(pipe_fd[1]);
                /* Set STDIN of the new process to the other end of the
 pipe */
                dup2(pipe_fd[0], 0);
                /* Start the nsupdate executable */
                execve(argv[0], &argv[0], envp);
        }

        /* In the parent, close the other end of the pipe */
        close(pipe_fd[0]);

        /* Send the command to the child (nsupdate exec) */
        write(pipe_fd[1], cmd, strlen(cmd);

        /* Wait for the child to exit */
        waitpid(pid, &wstatus, 0);
        /*
         * if (WIFEXITED(wstatus) == true &&
         *     WEXITSTATUS(wstatus) == 0) then
         * it means that the command was sent succesfully
         */

        return 0;
}
```

From the client, communication happens over UDP to the port 0x0D1F. We did a minimal implementation sending only the name of the destination application (the specification above allows to send the source AP name and the source AE name):

```
#define LISTEN_PORT htons(0x0D1F)

/* client UDP socket */
struct sockaddr_in l_saddr;
memset((char *)&l_saddr, 0, sizeof(l_saddr));
l_saddr.sin_family      = AF_INET;  /* IPv4 */
l_saddr.sin_addr.s_addr = local_ip; /* command line parameter */
l_saddr.sin_port        = 0;        /* system will bind to an available
 socket */

/* server UDP socket */
struct sockaddr_in r_saddr;
memset((char *)&r_saddr, 0, sizeof(r_saddr));
r_saddr.sin_family      = AF_INET;
r_saddr.sin_addr.s_addr = dst_ip_addr; /* received from DNS server *?
r_saddr.sin_port        = LISTEN_PORT; /* defined in specification */

/* send the message containing the ap_name */
sendto(fd, ap_name, ap_name_len, 0, (struct sockaddr *)
 &r_saddr, sizeof(r_saddr))
```

On the server side, the message is received and a random UDP port is created similarly as above and a reply is sent to the UDP port assigned to the client flow. At this point, the shim's minimal flow allocation completes and regular data transfer can begin.

8.5.5. Validation

We verified that these procedures are sound and work. In order to do this, we set up a client on IP address 192.168.126.34 and a server at ip address 192.168.126.36. We also installed a private DNS server (bind), as shown by the following traffic traces from wireshark.

```
server

/* send a query to the DDNS server for the application name (echo_server)
 */
1 192.168.126.36 157.193.215.138 DNS 57 Standard query 0x04d8 SOA echo-
server

/* DDNS responds that this application name is not registered */
2 157.193.215.138 192.168.126.36 DNS 98 Standard query response 0x04d8 No
 such name SOA echo-server SOA <Root>
```

```
/* DDNS update: register echo-server (which is a root name) at
 192.168.126.36 */
3 192.168.126.36 157.193.215.138 DNS 72 Dynamic update 0x01ea SOA <Root> A
 192.168.126.36

/* DDNS server responds that the name is registered */
4 157.193.215.138 192.168.126.36 DNS 45 Dynamic update response 0x01ea SOA
 <Root>

/* Client selects a UDP port (43817) and sends a flow allocation request.
 */
/* The datagram contains the name "echo-server" (11 characters) */
1 192.168.126.34 192.168.126.36 UDP 39 43817 → 3359  Len=11

/* The server selects a UDP port (44632) and echoes this back to inform
 the client */
2 192.168.126.36 192.168.126.34 UDP 39 44632 → 43817  Len=11

/* Flow allocation is now complete, all communication for this flow will
 happen between the allocated UDP ports */
/* The client application now knows the server port, and sends a message
 */
3 192.168.126.34 192.168.126.36 UDP 44 43817 → 44632  Len=16

/* The server sends a message back to the client */
4 192.168.126.36 192.168.126.34 UDP 44 44632 → 43817  Len=16
```

```
client
/* A flow allocation request arrived, query the DDNS server for the AP
 echo-server */
1 192.168.126.34 157.193.215.138 DNS 57 Standard query 0x7b1a A echo-
server

/* The DDNS server responds with a positive reply, (the name of the server
 is oceanus) */
/* The returned mapping is <echo-server, 192.168.126.36> */
2 157.193.215.138 192.168.126.34 DNS 109 Standard query response 0x7b1a A
 echo-server A 192.168.126.36 NS oceanus A 157.193.215.138

/* The remainder of the communication is the same as seen from the server
 side */

/* The client selects a UDP port (43817) and sends a flow allocation
 request to the server on UDP port 3359 (0x0D1F) */
```

```
1 192.168.126.34 192.168.126.36 UDP 39 43817 → 3359  Len=11

/* The server selects a UDP port (44632) and echoes this back to inform
 the client */
2 192.168.126.36 192.168.126.34 UDP 39 44632 → 43817  Len=11

/* Flow allocation is now complete, all communication for this flow will
 happen between the allocated UDP ports*/
/* The client application now knows the server port, and sends a message
 */
3 192.168.126.34 192.168.126.36 UDP 44 43817 → 44632  Len=16

/* The server sends a message back to the client */
4 192.168.126.36 192.168.126.34 UDP 44 44632 → 43817  Len=16
```

8.5.6. Conclusion

The above test shows that the concepts underlying these shims are implementable and can be used to develop a complete shim DIF. We found a race condition during the initial tests that stem from doing incomplete flow allocation over UDP that resulted in lost datagrams. The flow allocation can complete on the client before the server has completed its flow allocation steps. The packet (3) above could be dropped at the server as there is no flow associated with that UDP port yet. Our prototype implementation has solved this by immediately binding to the UDP socket when a flow allocation request is received (instead of when the server application accepts the flow), so packets will be queued in the UDP stack. If the server decides not to accept the flow, the socket is simply closed and that queue is dropped.

Because of the dynamic allocation of identifiers (SAPs in case of the LLC shim and UDP ports in case of the shim DIF over UDP) requires specifying the destination name for the application, we recommend implementing a full flow allocator as set out by RINA over these shim DIFs for any applications that require a stable environment to avoid unforeseen race conditions.

9. Summary and Conclusions

Deliverable D4.1 built on D2.1 and D2.2 (security requirements analysis and PRISTINE reference framework) and described the concepts and high-level design of security and reliability functions, mechanisms, and techniques. D4.2 provided initial developments of these functions to meet the security and resiliency requirements. D4.3 reports on the final work carried out for further enhancement of the security and resiliency functions from specification, implementation and verification and validation work to actual realise these functionalities. The components developed in WP4 have been delivered to WP6 for integration and experimentation purposes. Summary of the work reported in this deliverable are as follows:

Authentication of IPC processes

WP4 has adapted the cryptographic operations and information exchange performed by TLS (Transport Layer protocol) as an authentication policy for CACEP. This allows for the use of a well-known certificate-based authentication procedure in the DAF/DIF environments. The specification, implementation, validation and performance analysis of this type of authentication work is reported in this deliverable. It was observed that in comparison with no authentication, the RTT is moderately impacted by the use of TLS handshake and associated SDU protection policies but has inverse impact on throughput when the packet sizes are increased.

SDU Protection

The SDU Protection module is a part of the IPC Process (IPCP) data path. The SDU protection is applied on a per-port basis, allowing each unique flow to have different SDU protection policy or configuration. The implementation of SDU Protection, reported in D4.2 for PoC, modified the IRATI SerDes kernel module to add a limited functionality for protecting the transferred data against external threats. This deliverable has described further enhancement and implementation of the SDU Protection module as an independent kernel module (as opposed to be part of the SerDes module) connected to the RMT module, supporting three distinct policy sets that can be configured/replaced independently of each other. These policies are: SDU Protection Lifetime Limiting Policy Set, SDU Protection Cryptographic Policy

Set, and SDU Protection Error Check Policy Set. The details of the interfaces of the individual policy sets needed for development of custom implementations has been described in this document. Some experimental tests have been carried out to show the impact of using SDU protection policies on RTT and throughput.

Capability Based Access Control

Two scenarios for the use of CBAC have been specified in this deliverable; 1) the enrolment scenario, access control process when an IPCP communicates with another IPCP to join a DIF, 2) the control of the CDAP operations for access to RIB objects. The work on specification, implementation and integration of CBAC mechanism into IRATI stack has been reported in this deliverable. PoC results have also been given. Regarding access control policies, various security threats have also been identified and some countermeasure actions have been proposed to combat them.

Multi-Level Security (MLS)

D4.2 reported the design and specification of two MLS components: Communications security and Boundary Protection Components (BPC). In this deliverable, the focus has been put on three options for specification of BPC. The BPC at an AP option has been implemented in a real testbed using IRATI stack. It has been verified and tested in a number of networking scenarios. The results have been evaluated and showed the applicability of MLS in a RINA environment.

Key Management System

The Key Management System is a functional component of the Management DAF, used to create and manage the keys assigned to the IPC Processes. The standalone case for RINA key management has been considered in PRISTINE where some local information is accessible by IPCP and then following successful enrolment with the management DIF, access to an appropriate subset of key material is made possible. A number of use-case has been described to show the interactions between Key Management Agents and the Key Manager. In this deliverable, the key management system/function architecture has been given. Work will be continued on the specification and implementation of Key Management System and will be reported at the end of the project.

Security Threats

We provided a comprehensive report in D4.1 for threat identification, risk analysis, and proposed the security controls to put in place to

mitigate the possible security threats. The complexity of creating a testbed environment for verification of defined security controls in protecting RINA assets led us to consider alternative means of verification. We have considered formal verification technique and applied ProVerif tool for formal analysis of RINA security and RINASim for a demonstration of identified security threats. Applying a formal tool for verification of security mechanism enables us to determine the attack traces and verify the properties of security measures applied to mitigate the security threats associated with these attacks.

Resiliency and High Availability

PRISTINE investigated resiliency from three different viewpoints. First of all there is resiliency against network link and router failures, where a policy was designed, implemented and validated based on the Loop-Free Alternates solution for IP. Secondly, a more innovative approach is based on leveraging whatever-cast, a key feature of RINA that was only conceptually explained at the start of PRISTINE. WP4 has revised name-space management and made key contributions to the architecture in order to provide resiliency by multi-casting between routers, which is not immediately feasible in IP networks. This feature has been validated using the RINASim software. Thirdly, service and application resiliency has been explored through a load-balancing policy. Furthermore, WP4 has developed two new specification for shim DIFs. PRISTINE development of management infrastructure identified the need for shim DIFs to support parallel flows, which is not possible in the IRATI shim DIF over Ethernet 802.1Q. The concepts underpinning these shim DIFs have been validated as well, but full implementations of these components would not directly contribute to the programmability objectives and were considered out of scope.

In summary, we have advanced the security and resiliency aspects of RINA, specified, and developed functions, especially on the subjects identified above, implemented, conducted the PoC tests, and provided the modular security and resiliency components to the RINA stack and to WP6 for trails in PRISTINE use-case scenarios and beyond.

References

[8022] IEEE 802.2 LLC: (ISO/IEC 8802-2:1998), IEEE Standard for Information technology — Telecommunications and information exchange between systems—Local and metropolitan area networks — Specific requirements — Part 2: Logical Link Control

[8023] IEEE 802.3 Ethernet: IEEE Standard for Ethernet

[abac] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone, `` Guide to Attribute Based Access Control (ABAC) Definition and Considerations", NIST Special Publication 800-162, January 2014.

[benantar] M. Benantar, Ed., ``Access Control Systems – Security, Identity Management, and Trust Models", Springer Book, 2006.

[Boddapati2012] Boddapati, G., Day, J., Matta, I., & Chitkushev, L. (2012). Assessing the security of a clean-slate Internet architecture: Security as byproduct of decoupling different concerns. In Proceedings - International Conference on Network Protocols, ICNP. doi:10.1109/ICNP.2012.6459947

[cbac] J. Dennis and E. V. Horn, ``Programming Semantics for Multiprogrammed Computations", Communications of the ACM, vol. 9, no. 3, pp. 143-155, 1966.

[cbacIoT] J. Hernandez Ramos, A. Jara, L. Mar in, and A. Skarmeta, ``Distributed Capability based Access Control for the Internet of Things", Journal of Internet Services and Information Security (JISIS), vol. 3, no. 3/4, 2013.

[D2.1] 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_v1_0.pdf, accessed June 2016.

[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.

[D4.2] Hamid Asgari, Editor. PRISTINE Consortium. Deliverable 4.2. Initial specification and proof of concept implemenration of Innovative Security and Reliability Enablers. June 2015. Available online: http://ict-pristine.eu/wp-content/uploads/2013/12/pristine-d42-innovative-security-and-reliability-enablers_2_0.pdf, accessed June 2016.

[D53] PRISTINE D5.3 deliverable: http://ict-pristine.eu/wp-content/uploads/2013/12/pristine_d53-proof-of-concept-dms.pdf

[dcap] J. L. Hernandez-Ramos, A.J. Jara, L. Marin, and A. F. Skarmeta Gomez, ``DCapBAC: embedding authorization logic into smart things through ECC optimizations'', Int. J. Comput. Math. 93, 2, p. 345-366, February 2016.

[DeepSec] Deep Secure XML Guard Brochure, http://www.deep-secure.com/wp-content/uploads/2014/06/xml-guard-brochure1.pdf, accessed March 2016.

[Dolev1983] Dolev, D., & Yao, A. (1983). On the Security of Public Key Protocols. IEEE Transaction on Information Theory, 29(2), 198–208.

[Gollmann] D. Gollmann, “Computer Security”, Second Edition, John Wiley & Sons, November 2005.

[json] M. Jones, J. Bradley, and N.Sakimura. JSON Web Token (JWT). IETF Internet-draft (work in progress), July 2013. http://tools.ietf.org/html/draft-ietf-oauth-json-web-token-11.

[json-rfc] JWT RFC: https://tools.ietf.org/html/rfc7519

[jun-patent] R. M. Krohn, S. Ramamoorthi, M. Freed, K. Holleman. “Stateful firewall protection for control plane traffic within a network device”, Patent US7546635 B1, June 2009.

[KMIP] Key Management Interoperability Protocol Usage Guide Version 1.2. Edited by Indra Fitzgerald and Judith Furlong. Latest version: http://docs.oasis-open.org/kmip/ug/v1.2/kmip-ug-v1.2.doc

[Maven] Apache Maven, https://maven.apache.org/, accessed May 2016.

[Nexor] Nexor Watchman Datasheet, http://nexor.co.uk/sites/default/files/Nexor%20Datasheet%20-%20Nexor%20Watchman%207.0.pdf, accessed March 2016.

[NIST] NIST.SP800-57: Recommendation for key management (http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4)

[openssl] OpenSSL library: https://www.openssl.org/docs/manmaster/crypto/crypto.html

[rbac] D. Ferraiolo, J. Cugini, and R Kuhn, "Role based access control (RBAC): Features and motivations", in Proc. of 11th Annual Computer Security Application Conference, 1995, pp. 241.

[reach] J. Day, I. Matta, and K. Mattar. "Networking is IPC: A Guiding Principle to a Better Internet", in Proceedings of the 2008 ACM CoNEXT Conference, December 2008

[RFC826] RFC 826: Address Resolution Protocol. Online at http://tools.ietf.org/html/rfc826

[RFC5246] T. Dierks, E.Rescorla. "The Transport Layer Security (TLS) Protocol version 1.2". IETF Network Working Group, RFC 5246. August 2008.Available online: https://www.ietf.org/rfc/rfc5246.txt, accessed June 2015.

[rfc6192] D. Dugal, C. Pignatoro, D. Duhn. "Protecting the router control plane", IETF RFC 6192, March 2011.

[secicc2016] Eduard Grasa, Ondrej Rysavy, Ondrej Lichtner, Hamid Asgari, John Day, L. C. (2016). From protecting protocols to protecting layers: designing, implementing and experimenting with security policies in RINA. IEEE ICC 2016, Next-Generation Networking and Internet Symposium.

[Spring] Spring Framework, https://projects.spring.io/spring-framework/, accessed May 2016.

[Sybard] Sybard ® ICA Guard, Datasheet, QinetiQ, 2008. Available online: http://www.boldonjames.com/assets/downloadableFiles/sybard_ica_guard.pdf, accessed June 2015.

[Vesely2015] Vesely, V., Marek, M., Hykel, T., & Rysavy, O. (2015). Skip This Paper - RINASim: Your Recursive InterNetwork Architecture Simulator. In Proceedings of the "OMNeT++ Community Summit 2015 (pp. 1–5). Retrieved from http://arxiv.org/abs/1509.03550

[Vrijders2013]Sander Vrijders, Eleni Trouva, John Day, Eduard Grasa, Dimitri Staessens, Didier Colle, Mario Pickavet, Lou Chitkushev (2013) 5th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), Reliable Networks Design and Modelling workshop (RNDM), pp 215-221.

[xacml] XACML Specification: https://www.oasis-open.org/committees/download.php/4104/cs-xacml-specification-1.1.pdf

A. Traces of Authentication Verification Experiments

A.1. AuthNPassword Policy Traces

ARP request and response

```
09:48:33.467682 08:00:27:4d:bf:88 (oui Unknown) > Broadcast, ethertype
 802.1Q (0x8100), length 68:
 0x0000:  0001 d1f0 060f 0001 0800 274d bf88 7465  ..........'M..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                                     //
09:48:33.468124 08:00:27:48:d1:bf (oui Unknown) > 08:00:27:4d:bf:88 (oui
 Unknown), ethertype 802.1Q (0x8100), length 68:
 0x0000:  0001 d1f0 060f 0002 0800 2748 d1bf 7465  ..........'H..te
 0x0010:  7374 322e 4952 4154 492f 312f 2f08 0027  st2.IRATI/1//..'
 0x0020:  4dbf 8874 6573 7431 2e49 5241 5449 2f31  M..test1.IRATI/1
 0x0030:  2f2f                                     //
                                        //
```

M_CONNECT message

```
09:48:33.468707 08:00:27:4d:bf:88 (oui Unknown) > 08:00:27:48:d1:bf (oui
 Unknown), ethertype 802.1Q (0x8100), length 237:
 0x0000:  3200 0000 0000 0000 0100 0002 0001 0000  2...............
 0x0010:  0000 0040 00cf 0000 0000 0008 7310 0018  ...@........s...
 0x0020:  012a 0032 0038 0048 0050 0092 015f 0a20  .*.2.8.H.P..._..
 0x0030:  5053 4f43 5f61 7574 6865 6e74 6963 6174  PSOC_authenticat
 0x0040:  696f 6e2d 746c 7368 616e 6473 6861 6b65  ion-tlshandshake
 0x0050:  1201 311a 3808 d1a6 baba 0512 1ccb 53e1  ..1.8.........S.
 0x0060:  d4d5 75ad 60cc 0b82 2f66 3c65 59fc 7ea6  ..u.`.../f<eY.~.
 0x0070:  21bd 19f1 27e7 0b6e 461a 0454 4f44 4f22  !...'..nF..TODO"
 0x0080:  0454 4f44 4f32 0641 4553 3132 389a 0100  .TODO2.AES128...
 0x0090:  a201 0a4d 616e 6167 656d 656e 74aa 0101  ...Management...
 0x00a0:  31b2 010b 7465 7374 322e 4952 4154 49ba  1...test2.IRATI.
 0x00b0:  0100 c201 0a4d 616e 6167 656d 656e 74ca  .....Management.
 0x00c0:  0101 31d2 010b 7465 7374 312e 4952 4154  ..1...test1.IRAT
 0x00d0:  49da 0100 e001 019e e3a3 84              I..........
```

Server hello and server certificate messages

```
09:48:33.471209 08:00:27:48:d1:bf (oui Unknown) > 08:00:27:4d:bf:88 (oui
 Unknown), ethertype 802.1Q (0x8100), length 179:
 0x0000:  3200 0000 0000 0000 0100 0003 0001 0000  2...............
 0x0010:  0000 0040 0095 0000 0000 0008 0010 0c18  ...@............
```

```
 0x0020:  002a 0c53 6572 7665 7220 4865 6c6c 6f32  .*.Server.Hello2
 0x0030:  0c53 6572 7665 7220 4865 6c6c 6f38 0042  .Server.Hello8.B
 0x0040:  3532 330a 0131 10d1 a6ba ba05 1a1c 3414  523..1........4.
 0x0050:  6f8a 4bde 1234 31c4 30c2 7467 f0af b118  o.K..41.0.tg....
 0x0060:  2024 5d7b dd5c ab27 715d 2204 544f 444f  .$]{.\.'q]".TODO
 0x0070:  2a04 544f 444f 4800 5000 9201 020a 009a  *.TODOH.P.......
 0x0080:  0100 a201 00aa 0100 b201 00ba 0100 c201  ................
 0x0090:  00ca 0100 d201 00da 0100 e001 007b 28b0  .............{(.
 0x00a0:  c3                                       .
09:48:33.473784 08:00:27:48:d1:bf (oui Unknown) > 08:00:27:4d:bf:88 (oui
 Unknown), ethertype 802.1Q (0x8100), length 1121:
 0x0000:  3200 0000 0000 0000 0100 0003 0001 0000  2...............
 0x0010:  0000 0040 0043 0400 0000 0008 0010 0c18  ...@.C..........
 0x0020:  002a 1253 6572 7665 7220 4365 7274 6966  .*.Server.Certif
 0x0030:  6963 6174 6532 1253 6572 7665 7220 4365  icate2.Server.Ce
 0x0040:  7274 6966 6963 6174 6538 0042 d607 32d3  rtificate8.B..2.
 0x0050:  070a d007 3082 03cc 3082 02b4 a003 0201  ....0...0.......
 0x0060:  0202 0900 e368 e433 cc59 3673 300d 0609  .....h.3.Y6s0...
 0x0070:  2a86 4886 f70d 0101 0b05 0030 6631 0b30  *.H........0f1.0
 0x0080:  0906 0355 0406 1302 5350 3112 3010 0603  ...U....SP1.0...
 0x0090:  5504 080c 0942 6172 6365 6c6f 6e61 310e  U....Barcelona1.
 0x00a0:  300c 0603 5504 0a0c 0569 3263 6174 310e  0...U....i2cat1.
 0x00b0:  300c 0603 5504 0b0c 0558 2e35 3039 310f  0...U....X.5091.
 0x00c0:  300d 0603 5504 030c 0643 4152 4f4f 5431  0...U....CAROOT1
 0x00d0:  1230 1006 0355 0407 0c09 4261 7263 656c  .0...U....Barcel
 0x00e0:  6f6e 6130 1e17 0d31 3630 3331 3531 3830  ona0...160315180
 0x00f0:  3530 315a 170d 3139 3032 3037 3138 3035  501Z..1902071805
 0x0100:  3031 5a30 6831 0b30 0906 0355 0406 1302  01Z0h1.0...U....
 0x0110:  5350 3112 3010 0603 5504 080c 0942 6172  SP1.0...U....Bar
 0x0120:  6365 6c6f 6e61 3112 3010 0603 5504 070c  celona1.0...U...
 0x0130:  0942 6172 6365 6c6f 6e61 310e 300c 0603  .Barcelona1.0...
 0x0140:  5504 0a0c 0569 3263 6174 310b 3009 0603  U....i2cat1.0...
 0x0150:  5504 0b0c 0249 5431 1430 1206 0355 0403  U....IT1.0...U..
 0x0160:  0c0b 5465 7374 322e 4952 4154 4930 8201  ..Test2.IRATI0..
                                  ......
```

Client certificate, client key exchange and client certificate verify messages

```
09:48:33.475161 08:00:27:4d:bf:88 (oui Unknown) > 08:00:27:48:d1:bf (oui
 Unknown), ethertype 802.1Q (0x8100), length 1121:
 0x0000:  3200 0000 0000 0000 0100 0002 0001 0000  2...............
 0x0010:  0000 0040 0043 0400 0000 0008 0010 0c18  ...@.C..........
 0x0020:  002a 1243 6c69 656e 7420 4365 7274 6966  .*.Client.Certif
 0x0030:  6963 6174 6532 1243 6c69 656e 7420 4365  icate2.Client.Ce
 0x0040:  7274 6966 6963 6174 6538 0042 d607 32d3  rtificate8.B..2.
```

```
 0x0050:  070a d007 3082 03cc 3082 02b4 a003 0201  ....0...0.......
 0x0060:  0202 0900 e368 e433 cc59 3672 300d 0609  .....h.3.Y6r0...
 0x0070:  2a86 4886 f70d 0101 0b05 0030 6631 0b30  *.H........0f1.0
 0x0080:  0906 0355 0406 1302 5350 3112 3010 0603  ...U....SP1.0...
 0x0090:  5504 080c 0942 6172 6365 6c6f 6e61 310e  U....Barcelona1.
 0x00a0:  300c 0603 5504 0a0c 0569 3263 6174 310e  0...U....i2cat1.
 0x00b0:  300c 0603 5504 0b0c 0558 2e35 3039 310f  0...U....X.5091.
 0x00c0:  300d 0603 5504 030c 0643 4152 4f4f 5431  0...U....CAROOT1
 0x00d0:  1230 1006 0355 0407 0c09 4261 7263 656c  .0...U....Barcel
 0x00e0:  6f6e 6130 1e17 0d31 3630 3331 3531 3735  ona0...160315175
 0x00f0:  3733 345a 170d 3139 3032 3037 3137 3537  734Z..1902071757
 0x0100:  3334 5a30 6831 0b30 0906 0355 0406 1302  34Z0h1.0...U....
 0x0110:  5350 3112 3010 0603 5504 080c 0942 6172  SP1.0...U....Bar
 0x0120:  6365 6c6f 6e61 3112 3010 0603 5504 070c  celona1.0...U...
 0x0130:  0942 6172 6365 6c6f 6e61 310e 300c 0603  .Barcelona1.0...
 0x0140:  5504 0a0c 0569 3263 6174 310b 3009 0603  U....i2cat1.0...
 0x0150:  5504 0b0c 0249 5431 1430 1206 0355 0403  U....IT1.0...U..
 0x0160:  0c0b 5465 7374 312e 4952 4154 4930 8201  ..Test1.IRATI0..
                                ......

09:48:33.476260 08:00:27:4d:bf:88 (oui Unknown) > 08:00:27:48:d1:bf (oui
 Unknown), ethertype 802.1Q (0x8100), length 403:
 0x0000:  3200 0000 0000 0000 0100 0002 0001 0000  2...............
 0x0010:  0000 0040 0075 0100 0000 0008 0010 0c18  ...@.u..........
 0x0020:  002a 1343 6c69 656e 7420 6b65 7920 6578  .*.Client.key.ex
 0x0030:  6368 616e 6765 3213 436c 6965 6e74 206b  change2.Client.k
 0x0040:  6579 2065 7863 6861 6e67 6538 0042 8602  ey.exchange8.B..
 0x0050:  3283 020a 8002 d3a7 b332 6b1c 8d4e d89f  2........2k..N..
 0x0060:  c4d4 15f3 6b2c 0b20 fa5d aa3c 1e0f 307d  ....k,...].<..0}
 0x0070:  d1e1 805b 3ebc 6039 5ff7 c823 58a6 5ae9  ...[>.`9_..#X.Z.
                                ......

09:48:33.483746 08:00:27:4d:bf:88 (oui Unknown) > 08:00:27:48:d1:bf (oui
 Unknown), ethertype 802.1Q (0x8100), length 415:
 0x0000:  3200 0000 0000 0000 0100 0002 0001 0000  2...............
 0x0010:  0000 0040 0081 0100 0000 0008 0010 0c18  ...@............
 0x0020:  002a 1943 6c69 656e 7420 6365 7274 6966  .*.Client.certif
 0x0030:  6963 6174 6520 7665 7269 6679 3219 436c  icate.verify2.Cl
 0x0040:  6965 6e74 2063 6572 7469 6669 6361 7465  ient.certificate
 0x0050:  2076 6572 6966 7938 0042 8602 3283 020a  .verify8.B..2...
 0x0060:  8002 2f2e ce19 ce8e b9a5 bf77 ad62 fb7f  ../........w.b..
 0x0070:  e454 438e c8f0 f8d7 4b92 99cd ed3a 5283  .TC.....K....:R.
 0x0080:  8385 f35a b13d 63aa 4d43 c81c 9c29 e137  ...Z.=c.MC...).7
 0x0090:  1ee7 f82f 3db4 b683 da22 eb0a 5ca9 a5c7  .../=...."..\...
 0x00a0:  3e06 8f0f 3d3b 4a39 61d9 0be5 e608 991a  >...=;J9a.......
                                ......
```

Client and server Change Cipher Spec messages

```
09:48:33.484267 08:00:27:4d:bf:88 (oui Unknown) > 08:00:27:48:d1:bf (oui
 Unknown), ethertype 802.1Q (0x8100), length 155:
 0x0000:  3200 0000 0000 0000 0100 0002 0001 0000  2...............
 0x0010:  0000 0040 007d 0000 0000 0008 0010 0c18  ...@.}..........
 0x0020:  002a 1943 6c69 656e 7420 6368 616e 6765  .*.Client.change
 0x0030:  2063 6970 6865 7220 7370 6563 3219 436c  .cipher.spec2.Cl
 0x0040:  6965 6e74 2063 6861 6e67 6520 6369 7068  ient.change.ciph
 0x0050:  6572 2073 7065 6338 0042 0332 0160 4800  er.spec8.B.2.`H.
 0x0060:  5000 9201 020a 009a 0100 a201 00aa 0100  P...............
 0x0070:  b201 00ba 0100 c201 00ca 0100 d201 00da  ................
 0x0080:  0100 e001 004b 852d ae                   .....K.-.
09:48:33.486089 08:00:27:48:d1:bf (oui Unknown) > 08:00:27:4d:bf:88 (oui
 Unknown), ethertype 802.1Q (0x8100), length 155:
 0x0000:  3200 0000 0000 0000 0100 0003 0001 0000  2...............
 0x0010:  0000 0040 007d 0000 0000 0008 0010 0c18  ...@.}..........
 0x0020:  002a 1953 6572 7665 7220 6368 616e 6765  .*.Server.change
 0x0030:  2063 6970 6865 7220 7370 6563 3219 5365  .cipher.spec2.Se
 0x0040:  7276 6572 2063 6861 6e67 6520 6369 7068  rver.change.ciph
 0x0050:  6572 2073 7065 6338 0042 0332 0108 4800  er.spec8.B.2..H.
 0x0060:  5000 9201 020a 009a 0100 a201 00aa 0100  P...............
 0x0070:  b201 00ba 0100 c201 00ca 0100 d201 00da  ................
 0x0080:  0100 e001 00be dce0 1b                   .........
```

Encrypted client and server Finished messages

```
09:48:33.486938 08:00:27:4d:bf:88 (oui Unknown) > 08:00:27:48:d1:bf (oui
 Unknown), ethertype 802.1Q (0x8100), length 150:
 0x0000:  30e1 94ed f3ab 1211 e250 ee89 8b1d aa03  0........P......
 0x0010:  e482 2409 dbb2 b6b3 ac2a f0b4 6d7a e692  ..$......*..mz..
 0x0020:  4128 a3fd 2bc8 8094 a0cc 2032 c26a 2135  A(..+......2.j!5
 0x0030:  a964 0da4 a713 4f21 1e15 a9db 90dc 951f  .d....O!........
 0x0040:  65f3 8d2f e45a ddae 2a22 5b73 08a3 67f3  e../.Z..*"[s..g.
 0x0050:  9ea2 52eb e7ab f1e5 021a b12c 2e02 5943  ..R........,..YC
 0x0060:  b4c6 fc48 1bd7 d5ab f0d8 3a07 97f0 4a77  ...H......:...Jw
 0x0070:  37bb c498 a107 bea0 3970 2f82 dcc4 ab74  7.......9p/....t
 0x0080:  468f f972                                F..r
09:48:33.487753 08:00:27:48:d1:bf (oui Unknown) > 08:00:27:4d:bf:88 (oui
 Unknown), ethertype 802.1Q (0x8100), length 150:
 0x0000:  36c5 900b 4039 838c c51e 34c6 c1fe aaea  6...@9....4.....
 0x0010:  e482 2409 dbb2 b6b3 ac2a f0b4 6d7a e692  ..$......*..mz..
 0x0020:  0b2c 562f ebf5 600f 3255 00ac bdee e150  .,V/..`.2U.....P
 0x0030:  9cb4 e6a8 740d fcb5 1a30 14dc 7677 ff76  ....t....0..vw.v
 0x0040:  65f3 8d2f e45a ddae 2a22 5b73 08a3 67f3  e../.Z..*"[s..g.
 0x0050:  9ea2 52eb e7ab f1e5 021a b12c 2e02 5943  ..R........,..YC
```

```
 0x0060:  b4c6 fc48 1bd7 d5ab f0d8 3a07 97f0 4a77  ...H......:...Jw
 0x0070:  37bb c498 a107 bea0 3970 2f82 dcc4 ab74  7.......9p/....t
 0x0080:  b9bb 6fa8                                ..o.
```

IPCP test1.IRATI log

```
9094(1464768076)#librina.syscalls (DBG): Invoking SYS_writeManagementSDU
 (330)
9094(1464768076)#ipcp[2].rib-daemon (DBG): Sent CDAP message of size 188
 through port-id 2:
Opcode: 0_M_CONNECT
Abstract syntax: 115
Authentication policy: Policy name: PSOC_authentication-tlshandshake
Supported versions: 1;

Source AP name: test1.IRATI
Source AP instance: 1
Source AE name: Management
Destination AP name: test2.IRATI
Destination AP instance: 1
Destination AE name: Management
Invoke id: 1
Version: 1

9094(1464768076)#cdap (DBG): Waiting timeout 10000 to receive a connection
 response
9094(1464768076)#rib (DBG): Starting add object operation over
 RIB(0x20c7e30), of object(0x20cc3f0) with fqn: '/rmt/n1flows/
port_id=2' (parent '/rmt/n1flows')
9094(1464768076)#rib (DBG): Add object operation over RIB(0x20c7e30), of
 object(0x20cc3f0) with fqn: '/rmt/n1flows/port_id=2', succeeded. Instance
 id: '33'
9094(1464768076)#ipcp[2].routing-ps-link-state (DBG): flow allocation
 waiting for enrollment
9094(1464768076)#ipcp[2].rib-daemon (DBG): Received CDAP message from N-1
 port 2
Opcode: 12_M_WRITE
Object class: Server Hello
Object name: Server Hello
Scope: 0

9094(1464768076)#librina.syscalls (DBG): Invoking SYS_readManagementSDU
 (329)
9094(1464768076)#ipcp[2].rib-daemon (DBG): Received CDAP message from N-1
 port 2
Opcode: 12_M_WRITE
```

```
Object class: Server Certificate
Object name: Server Certificate
Scope: 0

9094(1464768076)#librina.syscalls (DBG): Invoking SYS_writeManagementSDU
 (330)
9094(1464768076)#ipcp[2].rib-daemon (DBG): Sent CDAP message of size 1072
 through port-id 2:
Opcode: 12_M_WRITE
Object class: Client Certificate
Object name: Client Certificate
Scope: 0

9094(1464768076)#librina.syscalls (DBG): Invoking SYS_writeManagementSDU
 (330)
9094(1464768076)#ipcp[2].rib-daemon (DBG): Sent CDAP message of size 354
 through port-id 2:
Opcode: 12_M_WRITE
Object class: Client key exchange
Object name: Client key exchange
Scope: 0

9094(1464768076)#librina.tls-handshake (DBG): Generated encryption key of
 length 16 bytes: 1fa1e09e799f5951fc368e61a11330e6
9094(1464768076)#librina.syscalls (DBG): Invoking SYS_writeManagementSDU
 (330)
9094(1464768076)#ipcp[2].rib-daemon (DBG): Sent CDAP message of size 366
 through port-id 2:
Opcode: 12_M_WRITE
Object class: Client certificate verify
Object name: Client certificate verify
Scope: 0

9094(1464768076)#librina.syscalls (DBG): Invoking SYS_writeManagementSDU
 (330)
9094(1464768076)#ipcp[2].rib-daemon (DBG): Sent CDAP message of size 106
 through port-id 2:
Opcode: 12_M_WRITE
Object class: Client change cipher spec
Object name: Client change cipher spec
Scope: 0

9094(1464768076)#librina.syscalls (DBG): Invoking SYS_readManagementSDU
 (329)
9094(1464768076)#ipcp[2].rib-daemon (DBG): Received CDAP message from N-1
 port 2
```

```
Opcode: 12_M_WRITE
Object class: Server change cipher spec
Object name: Server change cipher spec
Scope: 0

9094(1464768076)#ipcp (DBG): Requesting the kernel to update crypto state
 on port-id: 2
9094(1464768076)#librina.nl-manager (DBG): NL msg RX. Fam: 25; Opcode:
 42_UPDATE_CRYPTO_STATE_RESP; Sport: 0; Dport: 9094; Seqnum: 1464768070;
 Response; SIPCP: 2; DIPCP: 0
9094(1464768076)#librina.nl-manager (DBG): NL msg TX. Fam: 25; Opcode:
 41_UPDATE_CRYPTO_STATE_REQ; Sport: 9094; Dport: 0; Seqnum: 1464768070;
 Request; SIPCP: 2; DIPCP: 2
9094(1464768076)#librina.core (DBG): Added event of type
 41_UPDATE_CRYPTO_STATE_RESPONSE and sequence number 1464768070 to events
 queue
9094(1464768076)#ipcp[2].core (DBG): Got event of type
 41_UPDATE_CRYPTO_STATE_RESPONSE and sequence number 1464768070
9094(1464768076)#librina.tls-handshake (DBG): Encryption and decryption
 enabled for port-id: 2
9094(1464768076)#librina.syscalls (DBG): Invoking SYS_writeManagementSDU
 (330)
9094(1464768076)#ipcp[2].rib-daemon (DBG): Sent CDAP message of size 95
 through port-id 2:
Opcode: 12_M_WRITE
Object class: Client finish
Object name: Client finish
Scope: 0

9094(1464768076)#librina.syscalls (DBG): Invoking SYS_readManagementSDU
 (329)
9094(1464768076)#ipcp[2].rib-daemon (DBG): Received CDAP message from N-1
 port 2
Opcode: 12_M_WRITE
Object class: Server finish
Object name: Server finish
Scope: 0

9094(1464768076)#ipcp[2].enrollment-task-ps-default (INFO): Authentication
 was successful, waiting for M_CONNECT_R
9094(1464768076)#librina.syscalls (DBG): Invoking SYS_readManagementSDU
 (329)
9094(1464768076)#cdap (DBG): Connection response received
9094(1464768076)#ipcp[2].rib-daemon (DBG): Received CDAP message from N-1
 port 2
Opcode: 1_M_CONNECT_R
```

```
Abstract syntax: 115
Authentication policy: Policy name:
Supported versions:

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

9094(1464768076)#ipcp[2].enrollment-task (DBG): M_CONNECT_R cdapMessage
 from portId 2
9094(1464768076)#librina.ipc-api (DBG):
 IPCManager.getRegisteredApplications called
9094(1464768076)#librina.syscalls (DBG): Invoking SYS_writeManagementSDU
 (330)
```

B. Log files from MLS Verification Tests

B.1. Verification Test 1

BPC Application Log File

```
2016-06-03 02:42:42,200 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - REJECT:
 Destination is not cleared to receive HIGH data
2016-06-03 02:42:42,201 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU blocked.
2016-06-03 02:42:42,246 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - REJECT:
 Destination is not cleared to receive HIGH data
2016-06-03 02:42:42,248 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU blocked.
2016-06-03 02:42:42,260 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - REJECT:
 Destination is not cleared to receive HIGH data
2016-06-03 02:42:42,260 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU blocked.
2016-06-03 02:42:42,263 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - REJECT:
 Destination is not cleared to receive HIGH data
2016-06-03 02:42:42,264 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU blocked.
2016-06-03 02:42:42,273 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - REJECT:
 Destination is not cleared to receive HIGH data
2016-06-03 02:42:42,274 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU blocked.
2016-06-03 02:50:27,853 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Data is classified as LOW
2016-06-03 02:50:27,855 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node2.listener-1--
2016-06-03 02:50:27,858 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Data is classified as LOW
2016-06-03 02:50:27,859 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node2.listener-1--
```

```
2016-06-03 02:50:27,894 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Data is classified as LOW
2016-06-03 02:50:27,896 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node2.listener-1--
2016-06-03 02:50:27,907 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Data is classified as LOW
2016-06-03 02:50:27,908 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node2.listener-1--
2016-06-03 02:50:27,919 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Data is classified as LOW
2016-06-03 02:50:27,920 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node2.listener-1--
```

Low Receiving Application Log File

```
2016-06-03 02:50:27,887 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is low.
2016-06-03 02:50:27,892 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is low.
2016-06-03 02:50:27,928 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is low.
2016-06-03 02:50:27,940 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is low.
2016-06-03 02:50:27,953 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is low.
```

B.2. Verification Test 2

BPC Application Log File

```
2016-06-03 04:27:32,099 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Data is classified as LOW
```

```
2016-06-03 04:27:32,100 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node1.listener-1--
2016-06-03 04:27:32,136 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Data is classified as LOW
2016-06-03 04:27:32,138 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node1.listener-1--
2016-06-03 04:27:32,154 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Data is classified as LOW
2016-06-03 04:27:32,154 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node1.listener-1--
2016-06-03 04:36:11,572 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Data is classified as LOW
2016-06-03 04:36:11,573 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node1.listener-1--
2016-06-03 04:36:11,621 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Data is classified as LOW
2016-06-03 04:36:11,622 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node1.listener-1--
2016-06-03 05:04:00,445 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl -
 INDETERMINATE: Data classification exceeds sender's clearance.
2016-06-03 05:04:00,448 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU blocked.
2016-06-03 05:04:00,451 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl -
 INDETERMINATE: Data classification exceeds sender's clearance.
2016-06-03 05:04:00,452 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU blocked.
2016-06-03 05:04:00,456 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl -
 INDETERMINATE: Data classification exceeds sender's clearance.
2016-06-03 05:04:00,457 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU blocked.
2016-06-03 05:04:00,460 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl -
 INDETERMINATE: Data classification exceeds sender's clearance.
```

```
2016-06-03 05:04:00,461 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU blocked.
2016-06-03 05:04:00,465 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl -
 INDETERMINATE: Data classification exceeds sender's clearance.
2016-06-03 05:04:00,466 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU blocked.
```

High Receiving Application Log File

```
2016-06-03 04:27:33,535 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is low. 1
2016-06-03 04:27:33,536 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is low. 2
2016-06-03 04:27:33,538 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is low. 4
2016-06-03 04:36:12,703 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is low. 2
2016-06-03 04:36:12,710 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is low. 3
```

B.3. Verification Test 3

BPC Application Log File

```
2016-06-03 04:45:27,419 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Destination is cleared to receive HIGH data
2016-06-03 04:45:27,420 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node1.listener-1--
2016-06-03 04:45:27,457 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Destination is cleared to receive HIGH data
2016-06-03 04:45:27,458 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node1.listener-1--
```

```
2016-06-03 04:45:27,471 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Destination is cleared to receive HIGH data
2016-06-03 04:45:27,471 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node1.listener-1--
2016-06-03 04:45:27,480 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Destination is cleared to receive HIGH data
2016-06-03 04:45:27,481 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node1.listener-1--
2016-06-03 04:45:27,491 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Destination is cleared to receive HIGH data
2016-06-03 04:45:27,492 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node1.listener-1--
2016-06-03 04:58:57,502 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Data is classified as LOW
2016-06-03 04:58:57,503 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node1.listener-1--
2016-06-03 04:58:57,542 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Data is classified as LOW
2016-06-03 04:58:57,543 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node1.listener-1--
2016-06-03 04:58:57,555 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Data is classified as LOW
2016-06-03 04:58:57,556 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node1.listener-1--
2016-06-03 04:58:57,565 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Data is classified as LOW
2016-06-03 04:58:57,566 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node1.listener-1--
2016-06-03 04:58:57,575 [Thread-1] DEBUG
 com.thalesgroup.uk.trt.pristine.mls.HighLowPolicyEngineImpl - ACCEPT:
 Data is classified as LOW
```

```
2016-06-03 04:58:57,575 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.BpcSduProcessor - SDU allowed.
 Forwarding to destination: trt.rina.apps.node1.listener-1--
```

High Receiving Application Log File

```
2016-06-03 04:45:28,233 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is high.
2016-06-03 04:45:28,237 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is high.
2016-06-03 04:45:28,252 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is high.
2016-06-03 04:45:28,262 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is high.
2016-06-03 04:45:28,280 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is high.
2016-06-03 04:58:57,869 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is low.
2016-06-03 04:58:57,873 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is low.
2016-06-03 04:58:57,885 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is low.
2016-06-03 04:58:57,896 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is low.
2016-06-03 04:58:57,906 [Thread-1] INFO
 com.thalesgroup.uk.trt.pristine.mls.StringMessageSduProcessor - Received
 sdu: The security classification of this message is low.
```

C. Code for formal verification of security controls

C.1. rina.pvl

Source code of rina.pvl

```
001 type Application.
002 type Ipcp.
003 type Dif.
004 type Address.
005 type Host.
006 type Cdap.
007 type Pdu.
008 type Sdu.
009 type SduProtection.
010
011 event SduReceiveEvent(Address, Sdu).
012 event SduAcceptEvent(Address, Sdu).
013 event SduDropEvent(Address, Sdu).
014
015 event PduSendEvent(Address, Pdu).
016 event PduDropEvent(Address, Pdu).
017 event PduAcceptEvent(Address, Pdu).
018
019 event CdapSendEvent(Application, Cdap).
020 event CdapReceiveEvent(Application, Cdap).
021 event CdapDropEvent(Application, Cdap).
022
023 (* Creates a management PDU:
024 * dif - dif name.
025 * address - source address of this pdu in its dif.
026 * address - target address of this pdu in its dif.
027 * cdap - payload of the PDU, which consists of CDAP message.
028 *)
029 fun MakeManagementPdu(Dif, Address, Address, Cdap) : Pdu [data].
030
031 (* Creates a data transfer PDU:
032 * dif - dif name.
033 * address - source address of this pdu in its dif.
034 * address - target address of this pdu in its dif.
035 * sdu - payload of the PDU, which consists of SDU.
036 *)
037 fun MakeDataPdu(Dif, Address, Address, Sdu) : Pdu [data].
038 reduc forall dif:Dif, src:Address, dst:Address, sdu:Sdu;
 GetSdu(MakeDataPdu(dif,src,dst,sdu)) = sdu.
```

```
039 reduc forall dif:Dif, src:Address, dst:Address, sdu:Sdu;
 GetSourceAddress(MakeDataPdu(dif, src, dst, sdu)) = src.
040 reduc forall dif:Dif, src:Address, dst:Address, sdu:Sdu;
 GetTargetAddress(MakeDataPdu(dif, src, dst, sdu)) = dst.
041
042
043 (* Creates SDU from PDU and applies the specified sdu protection. *)
044 fun MakeDifSdu(SduProtection, Pdu): Sdu.
045 reduc forall sp:SduProtection, pdu:Pdu; GetPdu(sp, MakeDifSdu(sp,
 pdu)) = pdu.
046
047 (* Creates SDU from CDAP message and applies the specified sdu
 protection. *)
048 fun MakeDafSdu(SduProtection, Cdap): Sdu.
049 reduc forall sp:SduProtection, cdap:Cdap; GetCdap(sp, MakeDafSdu(sp,
 cdap)) = cdap.
050
051 (* Creates an ordinary IPC process:
052  * dif - DIF Name
053  * address - Process address
054  * channel - north bound channel
055  * channel - south bound channel
056  *)
057 fun MakeIpcProcess(Dif, Address, channel) : Ipcp [data].
058
059 (* Creates a Shim IPC process:
060  * dif - Shim DIF Name
061  * address - Process address
062  * channel - north bound channel
063  * channel - media pdu channel
064  *)
065 fun MakeShimProcess(Dif, Address, channel) : Ipcp [data].
066
067 reduc forall dif:Dif,address:Address,north:channel;
068     GetNorthChannel(MakeIpcProcess(dif , address, north)) = north;
069     forall dif:Dif,address:Address,north:channel;
070     GetNorthChannel(MakeShimProcess(dif , address, north)) = north.
071
072 reduc forall dif:Dif,address:Address,north:channel;
073     GetAddress(MakeIpcProcess(dif , address, north)) = address;
074     forall dif:Dif,address:Address,north:channel;
075     GetAddress(MakeShimProcess(dif , address, north)) = address.
076
077 reduc forall dif:Dif,address:Address,north:channel;
078     GetDif(MakeIpcProcess(dif , address, north)) = dif ;
079     forall dif:Dif,address:Address,north:channel;
```

```
080     GetDif(MakeShimProcess(dif , address, north)) = dif .
081
082 (* Creates Router IPC process:
083  * dif - Shim DIF Name
084  * address - Process address
085  * channel - south-east channel
086  * channel - south-west channel
087  *)
088 fun MakeRouterProcess(Dif, Address, channel, channel) : Ipcp [data].
089
090
091 (* Sends cdap message from the application using the specified
092  * IPCP and applies SDU Protection to cdap message before sending it
 to
093  * supporting IPCP
094  *)
095
 letSendCdap(application:Application,ipcp:Ipcp,sp:SduProtection,cdap:Cdap)=
096   let ch = GetNorthChannel(ipcp) in
097   let sdu = MakeDafSdu(sp, cdap)
098   in event CdapSendEvent(application, cdap);
099      out(ch , sdu ).
100
101 (* Receives any cdap message from the supporting IPCP protected
102  * by the specified SDU Protection. Drops the message if the
103  * verification of SDU protection fails.
104  *)
105
 letReceiveCdap(application:Application,ipcp:Ipcp,sp:SduProtection,ch:channel)=
106   in(GetNorthChannel(ipcp), sdu:Sdu);
107   let cdap = GetCdap(sp, sdu)
108   in event CdapReceiveEvent(application, cdap);
109      out(ch,cdap)
110   else event SduDropEvent(GetAddress(ipcp), sdu).
111
112 (*
113  * Sends SDU in a newly created PDU to the target address
114  * using underlying IPCP. It protects the data using specified
115  * SDU protection.
116  *)
117 letSendSdu(ipcp:Ipcp,sdu:Sdu,trg:Address,sp:SduProtection,ch:channel)=
118    let src = GetAddress(ipcp) in
119    let pdu = MakeDataPdu(GetDif(ipcp), src, trg, sdu)
120    in event PduSendEvent(src, pdu);
121       out(ch, MakeDifSdu(sp, pdu)).
122
```

```
123 (* Reads sdu from the specified channel, check sdu protection,
 extracts pdu and
124  * if PDU address is correct then forwards it to north channel. *)
125 letReceiveSdu(ipcp:Ipcp,sp:SduProtection,ch:channel)=
126   in (ch,sduIn : Sdu);
127   event SduReceiveEvent(GetAddress(ipcp), sduIn);
128   let pdu = GetPdu(sp, sduIn)
129   in (event SduAcceptEvent(GetAddress(ipcp), sduIn);
130      if GetTargetAddress(pdu) = GetAddress(ipcp)
131      then event PduAcceptEvent(GetAddress(ipcp), pdu);
132           out(GetNorthChannel(ipcp), GetSdu(pdu))
133      else event PduDropEvent(GetAddress(ipcp), pdu))
134   else event SduDropEvent(GetAddress(ipcp), sduIn).
```