Deliverable-6.1

First iteration trials plan for System-level integration and validation

Deliverable Editor: Miguel Angel Puente, Atos

Publication date:	31-March-2015
Deliverable Nature:	Report
Dissemination level (Confidentiality):	PU (Public)
Project acronym:	PRISTINE
Project full title:	PRogrammability In RINA for European Supremacy of virTualIsed NEtworks
Website:	www.ict-pristine.eu
Keywords:	RINA, Test, Plan, Integration
Synopsis:	This document describes the system-level integration plans and verification activities for the first iteration of PRISTINE development, covering the PRISTINE SDK, the RINA stack, PRISTINE specific policies, the Management Agent and the Network Manager.

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: Miguel Angel Puente, Atos
i2cat: Eduard Grasa, Leonardo Bergesio
atos: Miguel Angel
cn: Kewin Rausch, Roberto Riggio
tssg: Micheal Crotty
nxw: Vincenzo Maffione
fit-but: Ondrej Rysavy, Vladimir Vesely
iminds: Sander Vrijders

Disclaimer

Executive Summary

The purpose of the work carried out within the context of D6.1 has been to integrate the different software components developed by PRISTINE partners over the IRATI RINA implementation during the first iteration of the project, thus enabling the experimentation activities carried out by T6.2. PRISTINE has built over the basic IRATI stack in three main directions:

- Designing a Software Development Kit (SDK) to facilitate the programmability of the IRATI RINA implementation; enabling the development of pluggable policies that tailor the behavior of the different IPC Process components. The initial work on the SDK has been reported as part of deliverable [D23].
- Implementing a subset of the policies researched and simulated by WP3 and WP4, reported in deliverables [D32] and [D42] respectively. The policies implemented for the first iteration target congestion control, routing, forwarding, multiplexing, authentication and encryption.
- The development of a proof of concept DIF Management System (DMS), consisting in a Manager and a Management Agent for the IRATI stack. The work on the proof of concept DMS is described in deliverable [D53].

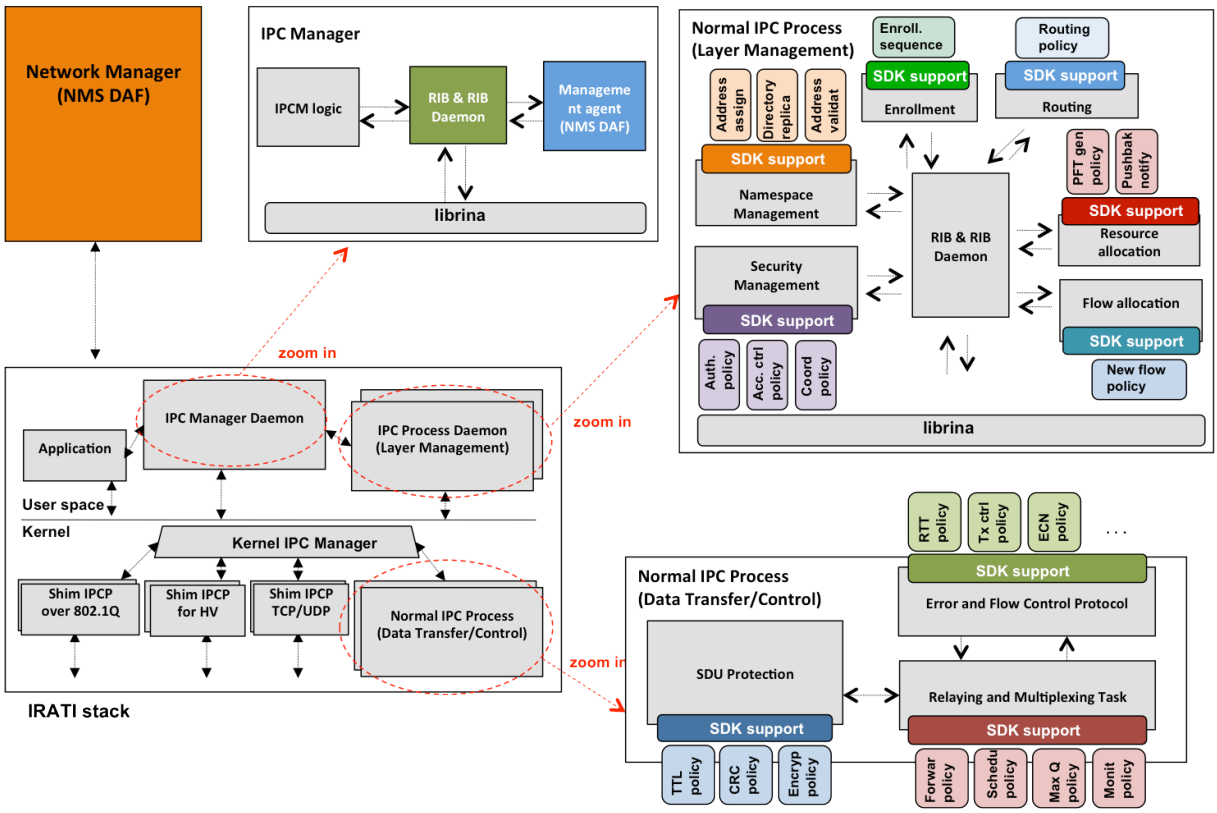

Figure 1. Basic IRATI stack, SDK, policies and Network Management System

These different software packages have been integrated on a use-case by use-case basis, producing three tested software bundles whose basic functionality has been verified at the system-level and are ready for its deployment in the experimental scenarios under development in T6.2. Figure 1 shows an overview of the basic IRATI stack and the software components developed by PRISTINE during the first iteration of the project. Grey boxes are components that were initially developed within the IRATI project - although they have been improved during the first iteration of PRISTINE - colored boxes highlight the work carried out by PRISTINE.

This report is structured as follows. Section 1 introduces the reference experimental facility that has been used for validation, as well as the different testbed configurations that will be used by T6.2 to carry out the experiments for each use case: distributed cloud, datacentre networking and network service provider. Section 2 provides an overview of the different policies to be integrated in the IRATI stack via the SDK, as well as the Network Manager and the Management Agent. After the description of the software components a discussion on how they will be integrated in the different experimental scenarios is provided, discussing any potential integration conflicts and the way to solve them. Next, section 3 describes the test plans designed by PRISTINE to validate the integrated software at the system level. Such test plans are divided into two main groups: i) common tests to check the correct behavior of the IRATI stack with the default policies and the Network Management System, and ii) specific test plans to exercise the specific policies designed for each one of the use cases. Finally section 4 presents the future plans for the T6.1 activity.

Table of Contents

1. Testbeds for Experimentation

To trial the project's outcomes in a real-life environment, PRISTINE can make use of facilities provided by different partners. Experimenting PRISTINE's results in real networks allows showing that these solutions can be used in current networks over IP, under IP and without IP. Task 2.1 produced a set of requirements that shall be tested, proving that they are appropriately fulfilled. To that end, PRISTINE aims to test the outcomes over real-world testbeds.

The testbeds considered by the project are the ones provided by iMinds (Virtual Wall), i2Cat (Experimenta), TSSG and Create-Net. PRISTINE will leverage on a unique facility, referred to as the "reference facility", to perform the experimental trials. This reference facility has been agreed to be the Virtual Wall testbed located in iMinds' premises [vwall]. The reasons that have led us to this choice are:

- **Controlled and automated environment**. Since all resources are co-located within the same physical location, reproducibility of experiments under the same or almost identical conditions becomes more feasible than in other distributed testbeds. Automation of deployment enables the scripting of experiments, which is key to effectively run a experiment multiple times.
- **Scale**. The Virtual Wall provides enough resources to be able to deploy and run all the experiments foreseen by PRISTINE.
- **Performance**. The Virtual Wall provides access to physical machines, thus minimizing the chances that virtualization becomes the limiting factor in any performance measurements carried out by experiments.

In the following sections first we describe the Virtual Wall testbed, and then we analyse the requirements posed by each of the use cases with respect to this testbed.

1.1. Reference facility: Virtual Wall

The Virtual Wall is an emulation environment that consists of 100 nodes (dual processor, dual core servers) interconnected via a non-blocking 1.5 Tb/s Ethernet switch, and a display wall (20 monitors) for experiment visualization. Each server is connected with 4 or 6 gigabit Ethernet links to the switch. The experimental setup is configurable through Emulab, allowing to create any network topology between the nodes, through VLANs on the switch. On each of these links, impairments (delay, packet loss, bandwidth limitations) can be configured. The Virtual Wall nodes can be assigned different functionalities ranging from terminal, server, network node to

impairment node. The nodes can be connected to test boxes for wireless terminals, generic test equipment, simulation nodes (for combined emulation and simulation) etc. The Virtual Wall features Full Automatic Install for fast context switching (e.g. 1 week experiments), as well as remote access.

Being an Emulab testbed at its core, the possibility to create any desired network topology and add any desired clients and servers makes it possible for the Virtual Wall to support a plethora of Future Internet experiments. These can focus more on networking experiments, e.g. emulating a large multi-hop topology in which advanced distributed load balancing algorithms are tested. However, the Virtuall Wall can just as well support experiments focusing on the application layer, e.g. performance testing of a novel web indexing service on a few server nodes, and creating a large amount of clients that feed it with a very large amount of data during the experiment. An illustration of this high level of flexibility displayed by the Virtual Wall is the fact that it is also applied in the context of OpenFlow experimentation in its role of a OFELIA island, and in the context of cloud experimentation in its role of a BonFIRE island.

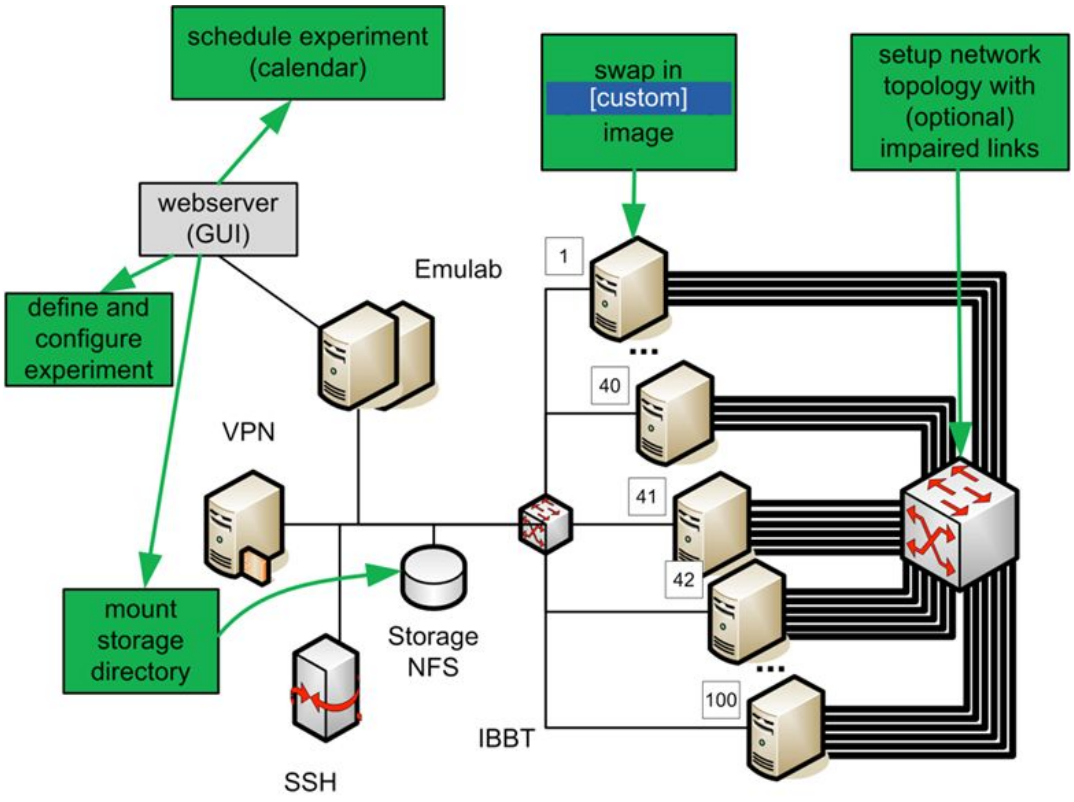

Figure 2. Virtual Wall topology

1.1.1. Virtual Wall deployments

There are currently 2 deployments of the Virtual Wall at iMinds (Virtual Wall 1 with 190 nodes (pcgen1 and pcgen2 nodes) and Virtual Wall 2 with 134 nodes (pcgen3 and

pcgen4 nodes)). In Fed4FIRE the initial target is to allow access to the newest instance, the Virtual Wall 2. An important characteristic of this instance is that all the nodes are also publically reachable through the public Internet using the IPv6 protocol.

The hardware can be used as bare metal hardware (operating system running directly on the machine) or virtualized through openVZ containers or XEN virtualization. XEN Virtualization comes into two flavours: using shared nodes (non-exclusive) where VMs are running on physical hosts which are shared with other experimenters, or using physical nodes (exclusive) which are exclusively used by your experiment. You have full control on all XEN parameters as you have root access to the DOM0. Network impairment (delay, packet loss, bandwidth limitation) is possible on links between nodes and is implemented with software impairment. Multiple operating systems are supported, e.g. Linux (Ubuntu, Centos, Fedora), FreeBSD, Windows 7. Some of the nodes are connected to an OpenFlow switch to be able to do OpenFlow experiments in a combination of servers, software OpenFlow switches and real OpenFlow switches.

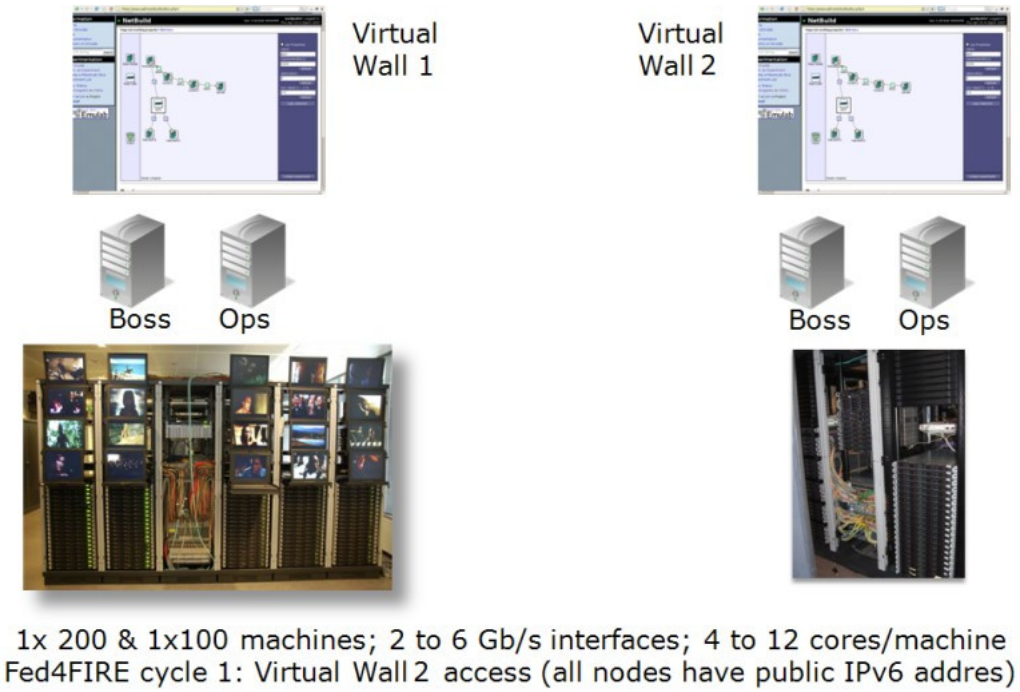

Figure 3. Virtual Wall deployments

The hardware of Virtual Wall 1 consists out of:

- 90x pcgen1 nodes: 2x Dual core AMD opteron 2212 (2GHz) CPU, 8GB RAM, 4x 80GB harddisk, 4-6 gigabit nics per node (20 nodes are wired to a display in the lab) (nodes n17x-xx)
- 100x pcgen2 nodes: 2x Quad core Intel E5520 (2.2GHz) CPU, 12GB RAM, 1x 160GB harddisk, 2-4 gigabit nics per node (nodes n14x-xx)

The hardware of Virtual 2 consists out of:

- 100x pcgen3 nodes: 2x Hexacore Intel E5645 (2.4GHz) CPU, 24GB RAM, 1x 250GB harddisk, 1-5 gigabit nics per node (nodes n095-xx, n096-xx, n097-xx)
- 6x pcgen1 nodes with 6 gigabit nics
- 28x pcgen4 nodes: 2x 8core Intel E5-2650v2 (2.6GHz) CPU, 48GB RAM, 1x 250GB harddisk, gigabit and 10gigabit connections (still to connect) (nodes n091-xx)
- 2x GPU nodes (n095-22, n097-22): 1x 6core Intel E5645 (2.4GHz), 12GB RAM, 4x 1TB disk in RAID5, 4x Nvidia Geforce GTX 580 (1536MB). Use the image urn:publicid:IDN+wall2.ilabt.iminds.be+image+dred:gpuready if you want to have a pre-installed SDK in /root

The following pcgen4 machines are equiped with RAID adapters and fast disks and are used as shared nodes, providing XEN VMs. It is also possible to have a public IPv4 address on VMs on these hosts (besides the default IPv6 public IP address):

- n091-01
- n091-06
- n091-26

1.1.2. Properties

Compared to the summary figure presented before, the situation at the Virtual Wall testbed is characterized by the following properties:

- Only Zabbix is supported as a monitoring framework (not Collectd nor Nagios).
- The depicted resources are Virtual Wall-specific: 100 dual processor, dual core servers, each equipped with multiple gigabit ethernet interfaces. These are all connected to one non-blocking 1.5 Tb/s Ethernet switch. The Virtual Wall system will automatically configure the corresponding VLAN settings in order to build any topology that the experimenter desires. In the example on the slide, the requested topology is that the first and last node should seem directly connected to each other on a LAN with a certain bandwidth and delay configuration. Therefore the system had automatically put an impairment node in between. However, from the perspective of the first and last node, this impairment node is invisible, and they believe that they are directly connected on layer 2.

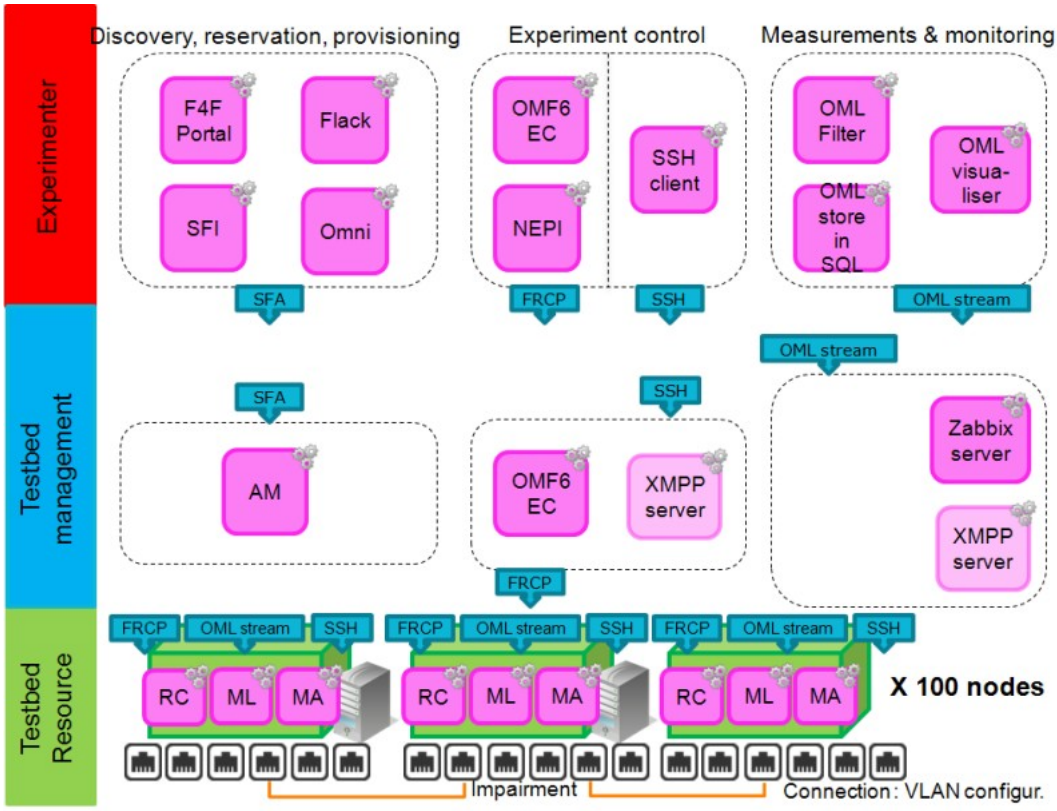

Figure 4. Virtual Wall properties

1.2. Use case requirements

The different use cases pose different requirements to the experimental testbed due to the different nature of the experiments dedicated to each of the use cases. In this section we analyse these requirements depending on each of the use cases' specifics. In principle these requirements are not focused on the reference facility, but on the underlying infrastructure that supports the experimentation. Therefore, for the experiments' success, the reference facility must comply with these requirements needed to achieve the desired evaluation results.

1.2.1. Distributed cloud

Experimental scenario to be demonstrated

Of the different approaches towards addressing the requirements of the VIFIB distributed cloud use case described in D3.2, WP6 trial activities will concentrate in the Figure below. In this scenario the resources in the VIFIB overlay support three types of DIFs: multiple tenant DIFs to isolate the applications belonging to different tenants; a cloud DIF, connecting together all the VIFIB resources that can instantiate user applications and a backbone DIF (with VIFIB systems just dedicated to relaying).

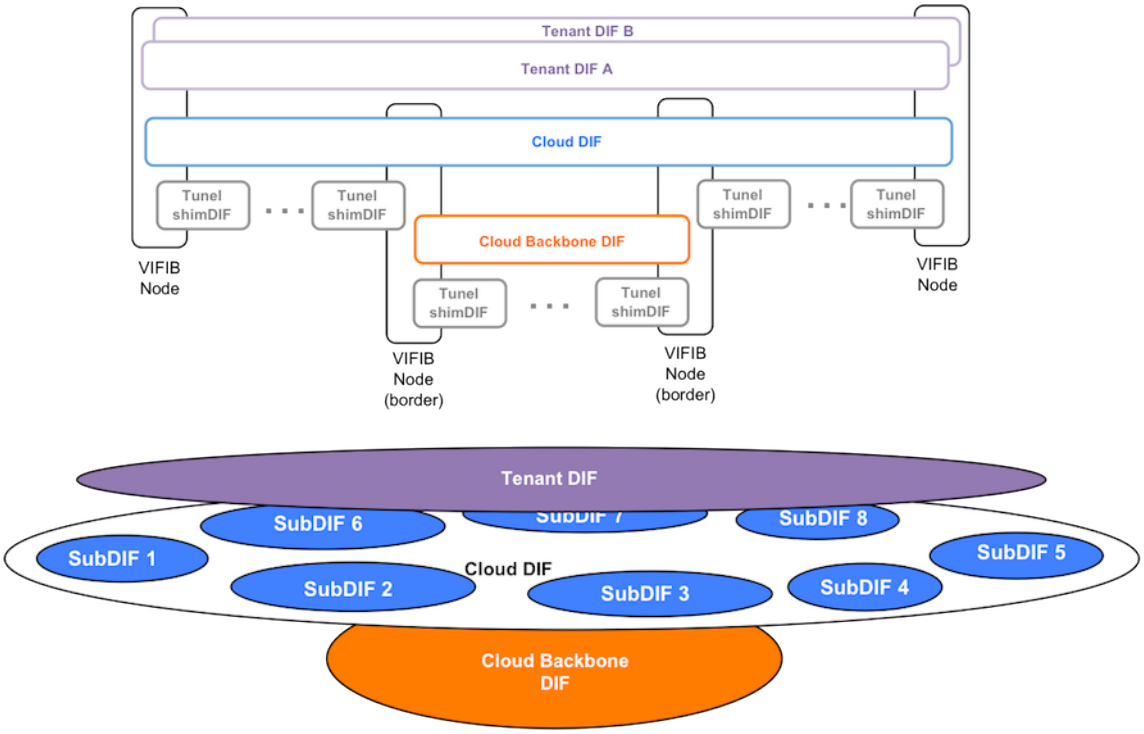

Figure 5. VIFIB distributed cloud scenario for experimentation activities

In general each tenant DIF will support a single application or set of applications of the same type, connecting together two or more instances of the application executing in different machines. Since the scope of the cloud DIF covers the whole overlay, there is no need for IPCPs in the tenant DIFs to relay, since all its IPC Process can be directly connected (no routing policy is required), as depicted in the Figure below. Since there is no relaying there is no need for congestion control policies in this DIF (end to end flow control is enough). Tenant DIFs can also have simple FIFO RMT policies and use different N-1 flows provided by the cloud DIF with different quality of service for applications with different loss/delay needs. Finally tenant DIF may use authentication with or without encryption policies depending on the requirements of the customer using those DIFs.

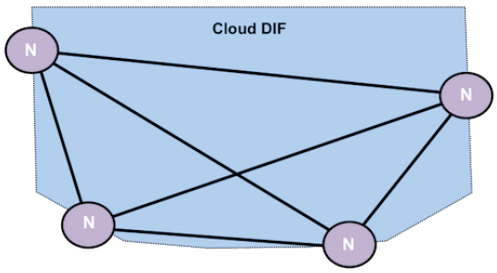

Figure 6. Example connectivity graph of a tenant DIF

The design of the cloud DIF is depicted in the Figure below. The cloud DIF connects together all the VIFIB nodes that can run tenant applications, which comprises most

but not all of the systems in the overlay (some of them are dedicated to supporting the backbone DIF, as it will be described later). The cloud DIF is divided into several sub-DIFs or "regions", corresponding to systems that are usually geographically close. Each region has two types of IPC Process: normal ones (labeled with an "N" in the Figure) and border routers (labeled with "BN" in the Figure). Border routers provide reliable inter-region connectivity via a full mesh of N-1 flows provided by the backbone DIF. This design allows all border routers of different regions to be one hope away from each other, eliminating the need to run an "inter-region" routing protocol and transferring the responsibility of routing between regions to the backbone DIF. Each region can internally run a routing policy such as link-state or distance-vector. As it will be explained later, it is important that the backbone DIF is overlaid over multiple ISPs, in order to maximize the reliability of the inter-region communication.

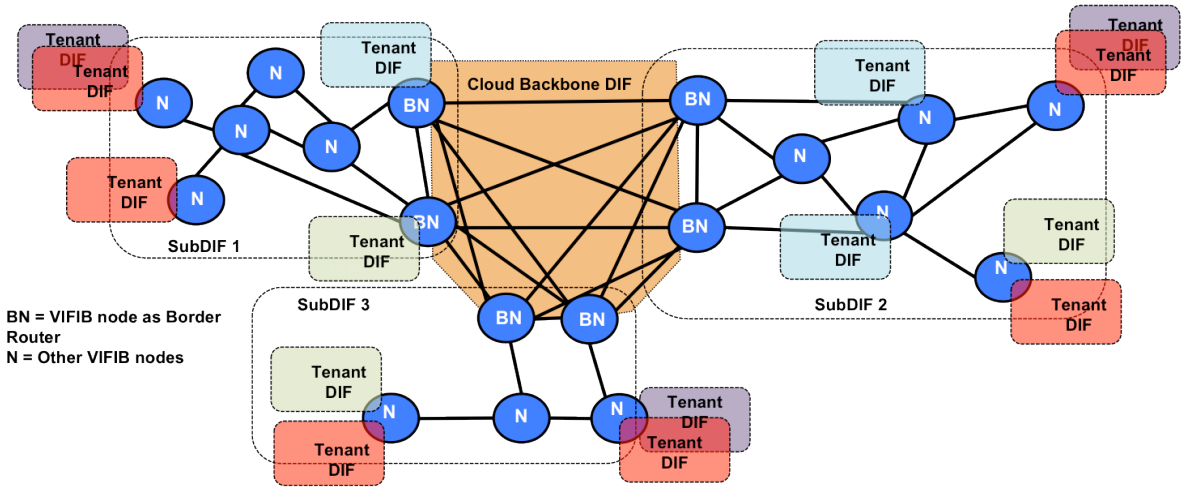

Figure 7. Design of the cloud DIF

The cloud DIF needs to support traffic differentiation, based on the different delay-loss requirements of the applications executing on the VIFIB overlay. An initial approach to this differentiation is to implement an RMT multiplexing policy that takes into account 4 traffic categories: i) low loss, low delay; ii) high loss, low delay; iii) low loss, high delay; and iv) high loss and high delay. Policies that deal with congestion control will also be necessary since the flows provided by the cloud DIF can span multiple hops; therefore end-to-end flow control is not enough to manage congestion.

The cloud DIF will be overlaid over Ethernet links, IP networks and the backbone DIF. Communications over the backbone DIF don't need to be authenticated nor encrypted, since both the cloud DIF and the backbone DIF are owned by VIFIB, joining the backbone DIF requires authentication and the backbone DIF will encrypt its own traffic before sending or receiving data over the public IP networks that are supporting it. Within a region (sub-DIF), IPCPs can be connected over public IP networks or over

Ethernets if IPCPs are running in physically collocated VIFIB nodes. In the former case authentication and encryption are mandatory, while in the latter it may not be necessary to mutually authenticate the IPCPs and encrypt the traffic exchanged over that N-1 flows.

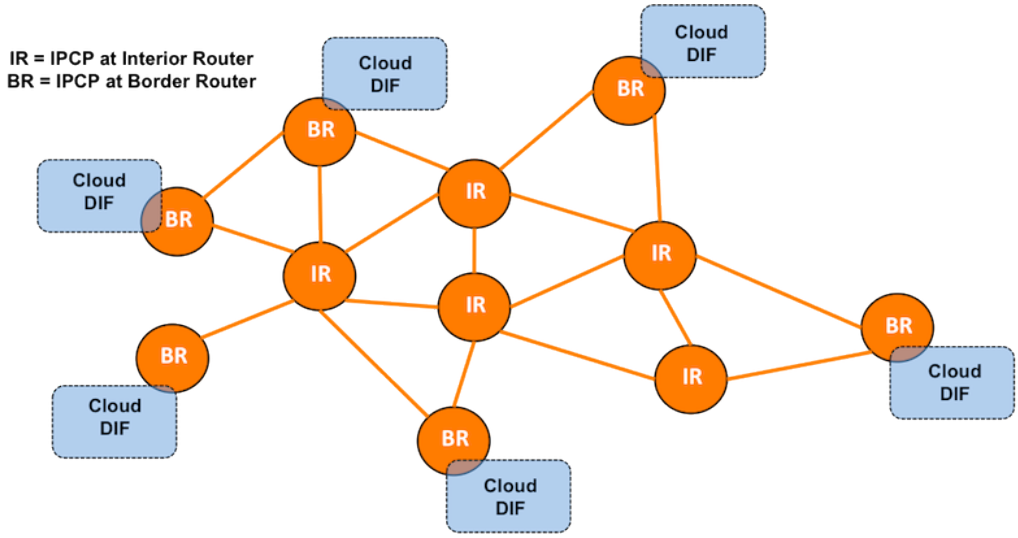

Figure 8. Design of the backbone DIF

The backbone DIF design is show in the Figure above. It is composed by VIFIB nodes dedicated to provide resilient inter-region connectivity to the different regions in the cloud DIF. The backbone DIF is overlaid on top of the public IP networks of several ISPs. It has to be designed to maximize the reliability of the flows provided by this DIF, not relaying on a single ISP. Therefore the connectivity within the backbone DIF should be as meshed as possible (without compromising the delay of the links). A resilient routing approach also needs to be considered, with quick detection of N-1 link failures and either fast re-routing and/or fast switching to pre-computed back-up paths to the next hop IPC Process.

Since traffic differentiation is already performed by the cloud DIF, the backbone DIF doesn't need to distinguish between different types of flows, therefore a simple FIFO scheduling strategy is sufficient to achieve the DIF goals. Congestion control policies are also required, since flows provided by the backbone DIF will typically span multiple hops. Authentication and encryption are also required for all the N-1 flows that support the operation of the backbone DIF, since it is overlaid over public networks of different ISPs.

Testbeds

Testbed 1: Virtual Wall

The final experimentation scenario, with all the required hardware and its configuration is shown in the figure below.

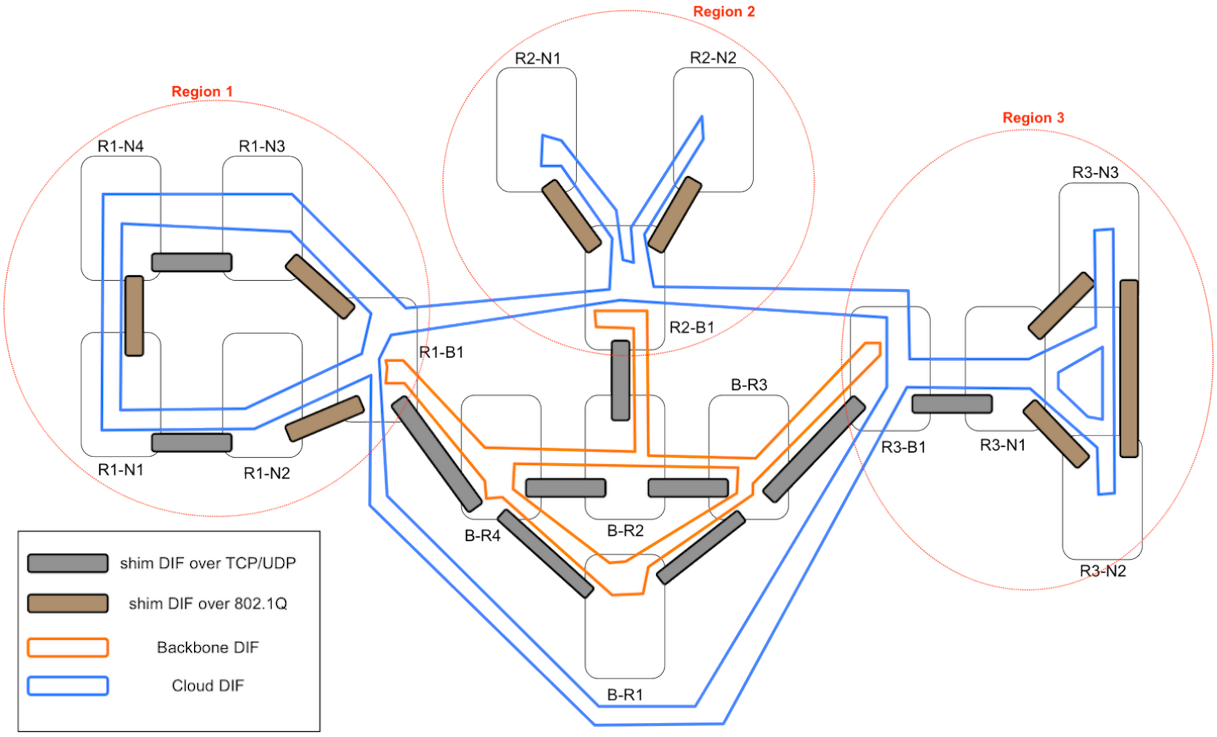

Figure 9. Testbed for the distributed cloud use case

16 systems have been arranged in 3 separate regions and the backbone: region 1 has 5 systems, region 2 has 4 systems and region 3 has 3 systems. Each regions has one border router, which connects to one of the 4 backbone interior routers. Backbone routers are interconnected via shim DIFs over TCP/UDP between them and to region border routers; while region routers are mostly connected via a shim DIF over 802.1Q and also via shim DIFs over TCP/UDP. The experimental scenario, while still manageable in size, should be large and rich enough to experiment with the different policies investigated in this use case.

Testbed 2: Nexedi Infrastructure

Once the scenario has been trialed in the Virtual Wall and validated via multiple experiments, the same configuration depicted in the former section can be deployed over the real Nexedi infrastructure, in order to repeat the experiments and compare the results with the ones obtained in the Virtual Wall controlled environment.

1.2.2. Datacenter networking

DatNet requirements from D2.1

In order to pave the way for the RINA implementation in the datacenter networks, the datacenter network requirements obtained in [D2.1] and mentioned in the following for convenience are:

- Uniform high capacity: The capacity between any two servers within a DC should be limited only by their Network Interface Card (NIC) transmission rate. The topology link's capacity should not limit the maximum intra-datacenter traffic rate. This is a topological issue, but the RINA implementation should be focused on improving this aspect. For example, using novel routing strategies that optimize the intra-DC traffic rate without the need of overpopulating the upper layers with additional links.
- Performance isolation: The QoS provided by a service should be unaffected by other services' traffic. That involves aspects such as flow isolation and resource reservation that RINA must fulfill accordingly. Resource reservation should be decoupled from "physical" resources, e.g. a certain ensured bandwidth between two nodes should not involve any specific path between them.
- Ease of management: A DC infrastructure should be easy to manage. Management aspects that should be addressed are:
  - "Plug-&-Play". The connection of new servers and nodes should be managed automatically by the network. Not special configuration must be needed before deployment.
  - Dynamic addressing. Any server or node's address/name must be dynamically assigned and managed by the network.
  - The server configuration should be the same as in a LAN.
  - Topology changes should be efficiently disseminated. Routing mechanisms should be highly convergent.
  - Legacy applications depending on broadcast must work.
- VM mobility: Any VM should be migrated to any physical machine within the DC infrastructure or to an associated external datacenter without a change in its name/address. In the same way, routing towards the new destination should converge seamlessly allowing new connections to the new destination in a transparent way.
- Fault Tolerance: Potential failures should be detected, addressed and solved efficiently. Typical failures such as forwarding loops, etc. should be addressed.

- Security: A RINA DC should allow tenants to configure security controls for their applications and it should offer secure, simple, uniform interfaces for configuring security policies and mechanisms. This should make it easier to configure security controls, reducing the likelihood of configuration errors that lead to security vulnerabilities. Cryptographic protection should be used to separate data on internal networks and to protect control and management information exchanges with distributed facilities as part of RINA Management plane (e.g. name, DIF and network management).

DatNet DIF configuration

The basic DIF configuration for the Datacenter Networking use case is depicted in the next figure, where:

- Datacentre fabric DIF: Connects together all the hosts in the DC, providing homogeneous bandwidth, non-blocking, etc. Uses a leaf-spine or multi-staged clos network topology (as show on previous slide). It allows the provider to allocate DC resources efficiently based on the needs of its tenants. Could be broken down in more DIFs.
- Tenant DIFs: Connects together all the VMs specified by a tenant, isolating them from other tenants or from other DIFs of the same tenant. Can also connect out of the datacenter (to customers or other sites of the tenant or to other DCs), for example over the public Internet (as shown in the Figure). In a sense, with this DIF the provider's domain becomes an independent unit of resource allocation.

This configuration will be the one implemented by the DatNet experiments, so it represents the base guideline for the analysis of the use case requirements.

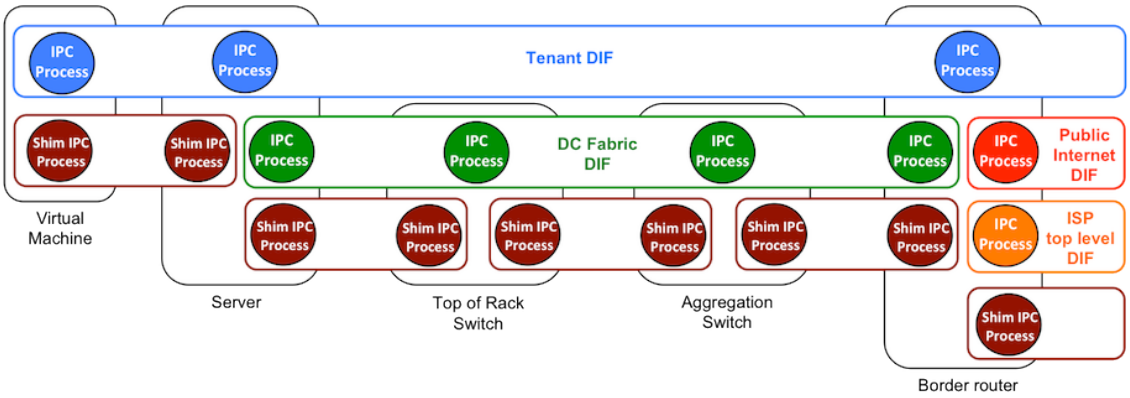

Figure 10. RINA applied to the DC use case

Topological requirements

The network topology to be experimented in the DatNet use case is the one represented in the following figure. The dashed blocks represent the so called "pods", which interconnect the servers (lower layer) to the core routers (upper layer).

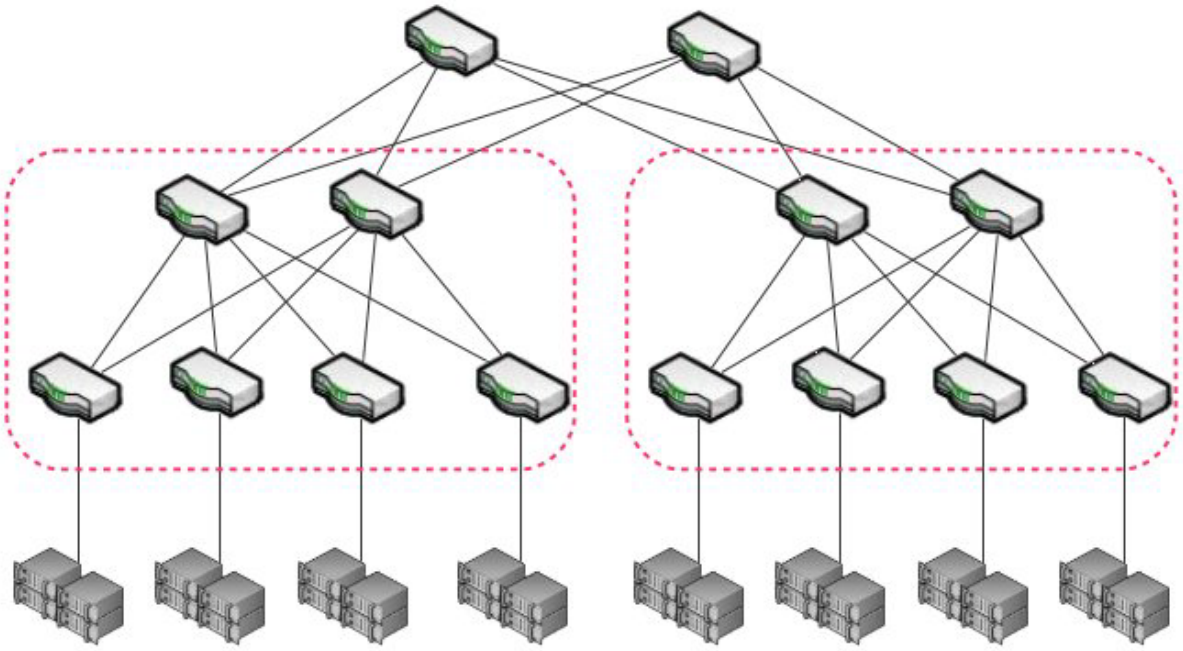

Figure 11. DatNet network topology

The requirements that the DatNet use case pose with respect the topology are the following:

- **Node interconnections**. The node interconnections must follow the topology of the previous figure. This means that the testbed must support nodes with multiple links to other nodes of the topology.
- **Number of interconnected nodes**. The number of nodes is a key aspect, since the topology envisaged for the DatNet use case is dependent on the number of servers. Therefore, to experiment with a given number of servers, the testbed must be able to allocate the necessary router nodes.
- **Number of links**. In the same way, the number of links of the topology is dependent on the number of servers, so the testbed must allow to allocate multiple links among the nodes.
- **Node and link characteristics**. In the topology considered for the DatNet use case, all the links have the same capacity. These links interconnect two kind

of physical nodes: routers and servers. In turn, the server nodes allocate virtual machines, which can be considered as logical nodes within the physical server nodes.

Traffic requirements

Apart from the topological requirements, the traffic requirements are also a key part for the experiments' success. These traffic requirements are the following:

- **Traffic sources and sinks**. The nodes acting as traffic sources and sinks are the core routers and the servers. The core routers act as entry and exit points of the datacenter traffic, and the servers are the destination of the incoming traffic or the source of the outgoing traffic. More specifically, the nodes within the server that act as sources or sinks are the VM nodes, rather than the physical server nodes which are the nodes that allocate the VMs.
- **Traffic load**. The experimentation with the proper traffic load is a key aspect for the correct experimental outcomes. The data center traffic can be divided in two classes:
  - **Extra-datacenter traffic**. The traffic that enters or leaves the datacenter. This traffic traverses all the datacenter network layers from the core routers to the servers or vice-versa.
  - **Intra-datacenter traffic**. The traffic that is conveyed between two datacenter nodes and do not leave the datacenter network. This traffic is proven to account for percentages up to the 80% of the total datacenter traffic as studied in [PRISTINE-D21].
- **Traffic characteristics**. The traffic generated at the traffic sources must follow a realistic pattern for the given application that is desired to be tested. The traffic may be transmitted in bursts, continous flows, etc.
- **Routing**. The datacenter networks employ routing strategies for load balancing purposes. There is not fixed paths between a certain pairs of nodes, instead the traffic is forwarded through different available paths following strategies such as Equal Cost Multipath routing (ECMP). Pristine is also investigating enhanced multipath routing mechanisms that shoulld be tested experimentally [PRISTINE-D31].
- **Congestion**. In current datacenter networks, congestion problems are handled by mechanisms such as Multipath TCP (MPTCP). Pristine is also investigating novel congestion control mechanisms, which shall be also tested accordingly.

Testbed: Virtual wall

The overview of the experiment for the DatNet use case over the Virtual Wall testbed is depicted in the following figure. The experiment overview consists on a simple topology where the shim DIFs connect the network nodes and VMs. The overview depicts two tenant DIFs for two different tenants with two VMs each.

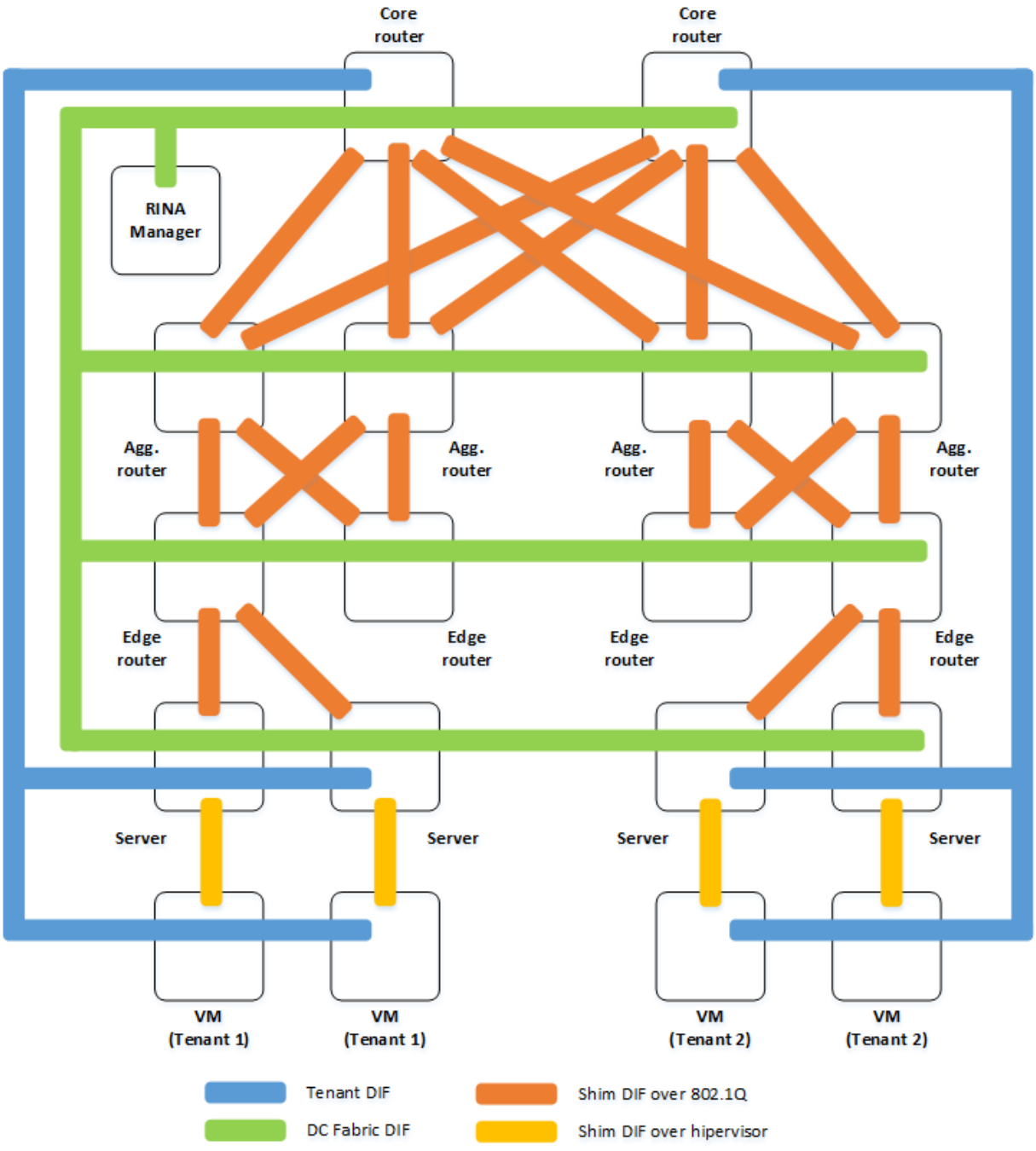

Figure 12. DatNet experiment overview

1.2.3. Network service provider

Experimental scenario to be demonstrated

The figure below shows an overview of the Service chaining experiment. The classifier must separate input traffic in different IP flows (using information in the IP or TCP/ UDP address headers). Each flow must be routed by the SFFs (Service Function Forwarders) in a different way, going through a series of functions. The orchestrator configures the Classifier and SFFs directly, and then instructs OpenStack to instantiate the right Functions in a number of VMs, and configuring the networking to plug them into the appropriate chains.

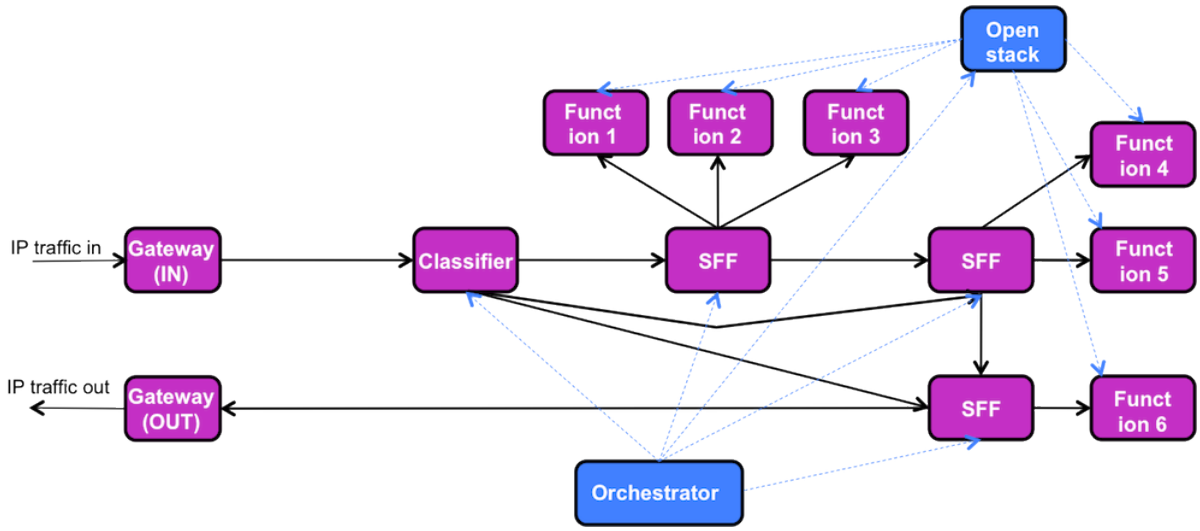

Figure 13. Service chaining experiment overview

A more detailed overview of the data flow through the service chains is provided in the figure below. The classifier separates the incoming input traffic in N different flows, each of which will be assigned to a separate service chain (the figure shows an example with two service chains: orange and green). Each service chain is composed by an SFF instance plus the instances of all the functions required in the chain. The classifier forwards the traffic of each flow to the proper SFF, which causes the IP flow to be processed by the functions instances in the right order (function instances just get traffic from the SFF, apply the function to the traffic, and forward the result back to the SFF). When the packets of the flow have gone through all the functions, the SFF routes the traffic to the output gateway (where the traffic exiting the service chain is aggregated back).

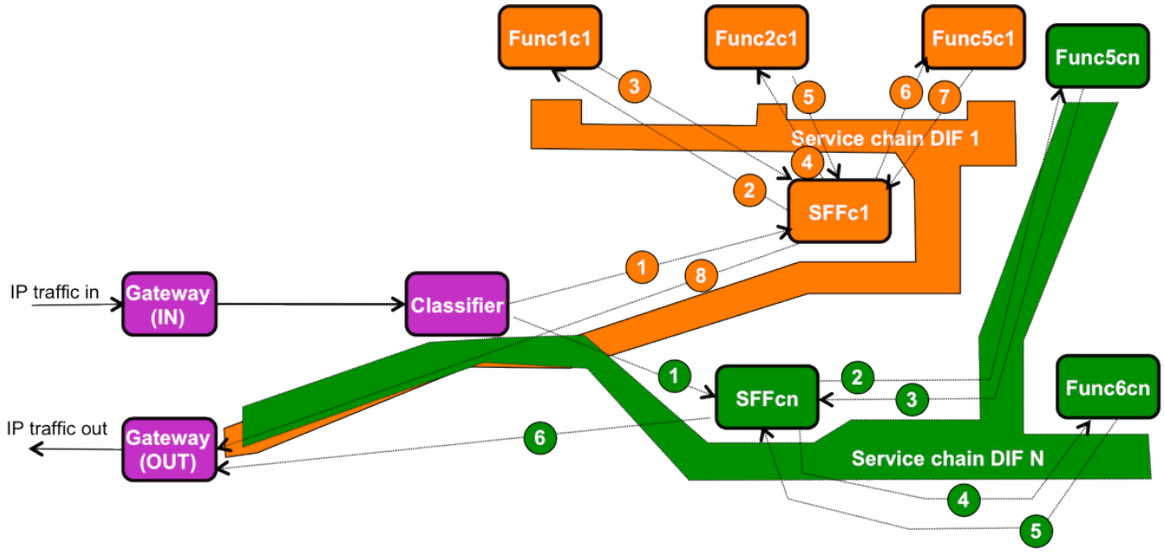

Figure 14. Service chaining with RINA deployed everywhere

In this scenario the orchestrator has to the the following configuration per chain: create the service chain DIF, create the SFF and function instances (via Openstack), configure the different instances to register to the service chain DIF. This environment provides the cleanest solution deploying RINA everywhere, but it is also the most costly for the project in terms of implementation, since the different functions (firewalls, NATs, etc) would have to be modified to send and receive traffic using RINA flows.

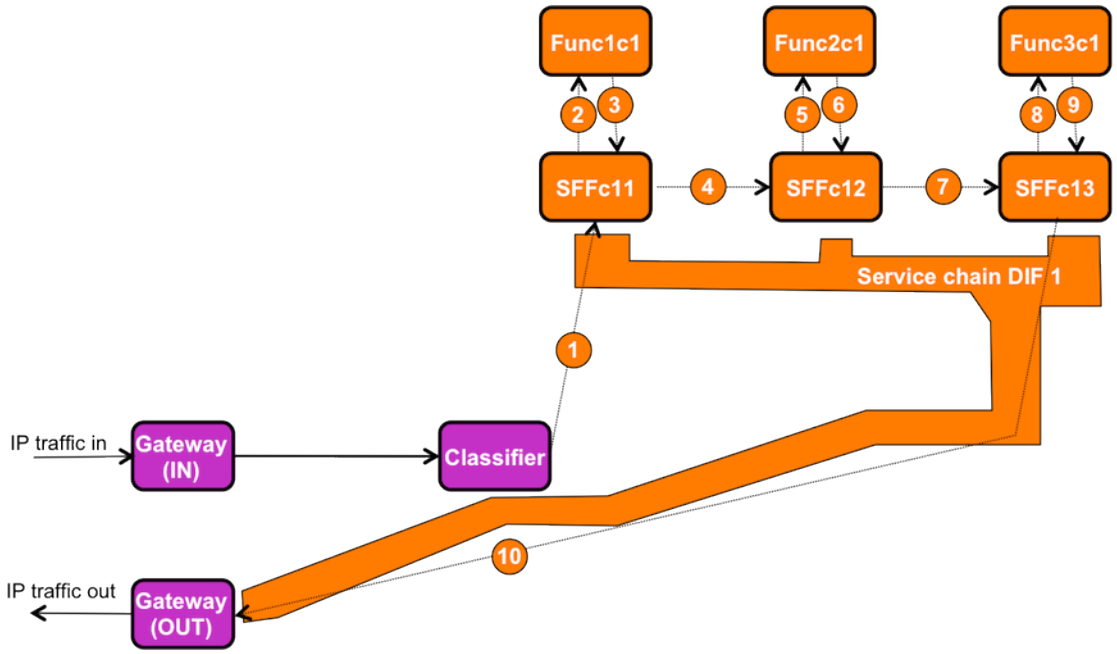

Figure 15. Service chaining with functions not supporting RINA

A good compromise in terms of functionality and implementation effort is shown in the figure above. In each chain there is one SFF instance per physical machine where a function VM is instantiated. Instances on the same physical machine can be connected

to its SFF via an internal VLAN. SFF instances know when to forward the data to the instance within the physical machine via the VLAN, or send the data to the next SFF instance of the chain via the Service chain DIF. This way we avoid modifying the service functions and we confine the scope of each VLAN to a single physical machine (minimizing the VLAN complexity setup and avoiding scalability limitations).

Testbed: Virtual Wall

The final experimentation scenario, with all the required hardware and its configuration is shown in the figure below.

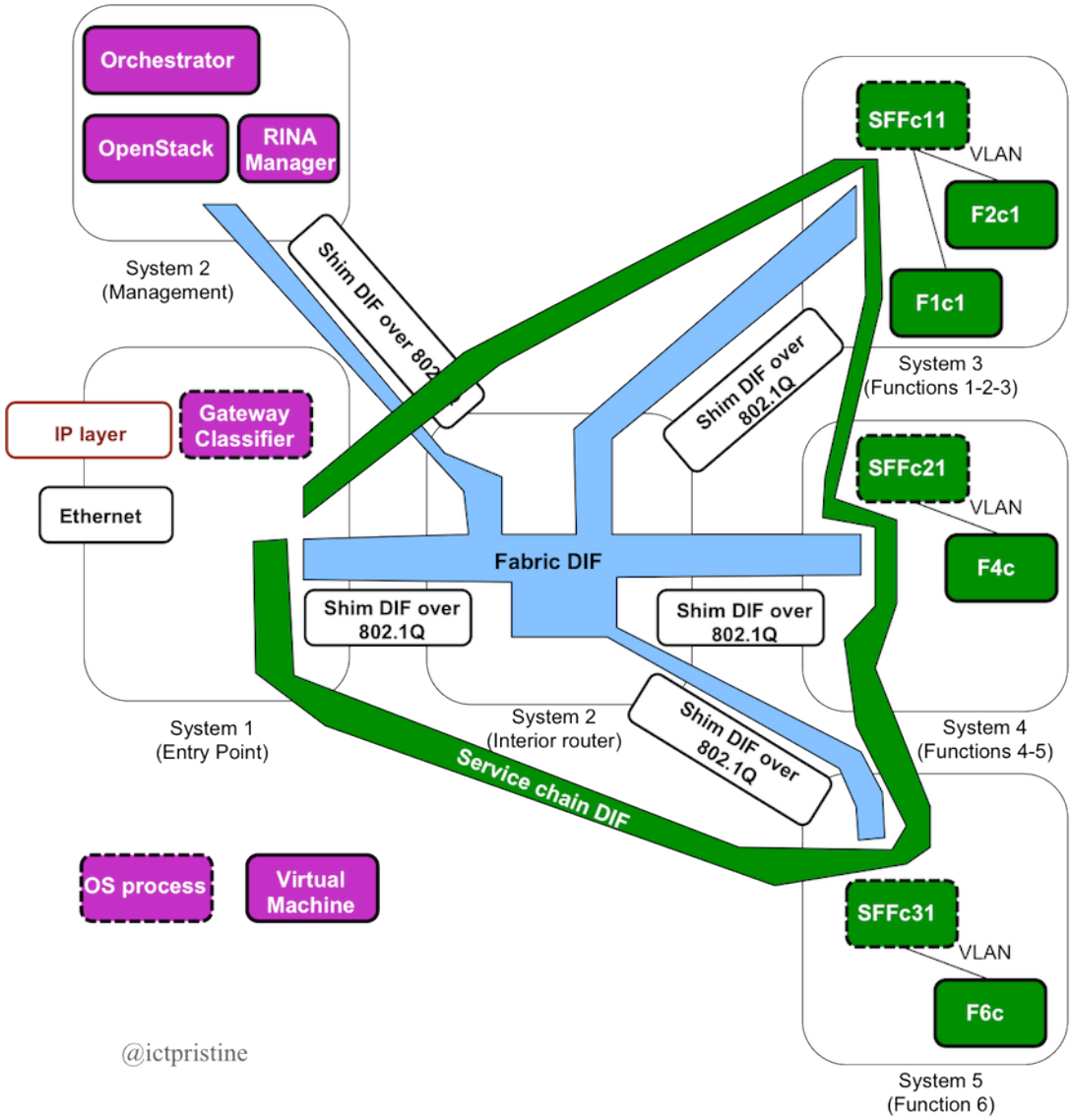

Figure 16. Service chaining experiment overview

System 1 is the entry point, where the gateways and classifier are located (for simplicity they can all be collocate in a single OS process). System 2 is dedicated to host the different Management software required to manage the service chaining

facility: OpenStack to instantiate VMs running specific functions and tie them to the appropriate SFF instance via a VLAN; the RINA Manager to configure and manage the DIFs supporting the service chain and the Orchestrator to coordinate the overall process. System 3 acts as a dedicated interior router that connects together all the other systems. Systems 4 to 6 are the ones running the actual functions (packaged as Virtual Machines). Each system is capable of instantiating a certain type of functions. The facility DIF causes all systems to be one hop away, facilitating Management and minimizing the number of IPC Processes required in the different service chain DIFs. Service chain DIFs float on top of the facility DIFs, and enable the communication of the Gateway/Classifier with the different SFF instances of a the service chains.

2. Integration plan

PRISTINE implements software for the different network fields to be implemented in the RINA stack. WP3 addresses congestion control, resource allocation and topological routing. WP4 addresses security, authentication and resiliency. And WP5 addresses multi-layer security, performance and configuration management. The different PRISTINE's partners are involved in separate implementations of the different WPs, so the integration effort is critical for achieving the final PRISTINE's outcome.

The next figure depicts the overview of the integration strategy. The different use cases will create software bundles to be trialed in T6.2. The objective of this section is to collect the contributions from WP3, WP4 and WP5 with respect to each of the use cases and analyse the integration to identify potential points of conflict. Later on, in section 4, we will define the integration test plans to be performed over the integration of the differnt parts provided by the PRISTINE's partners.

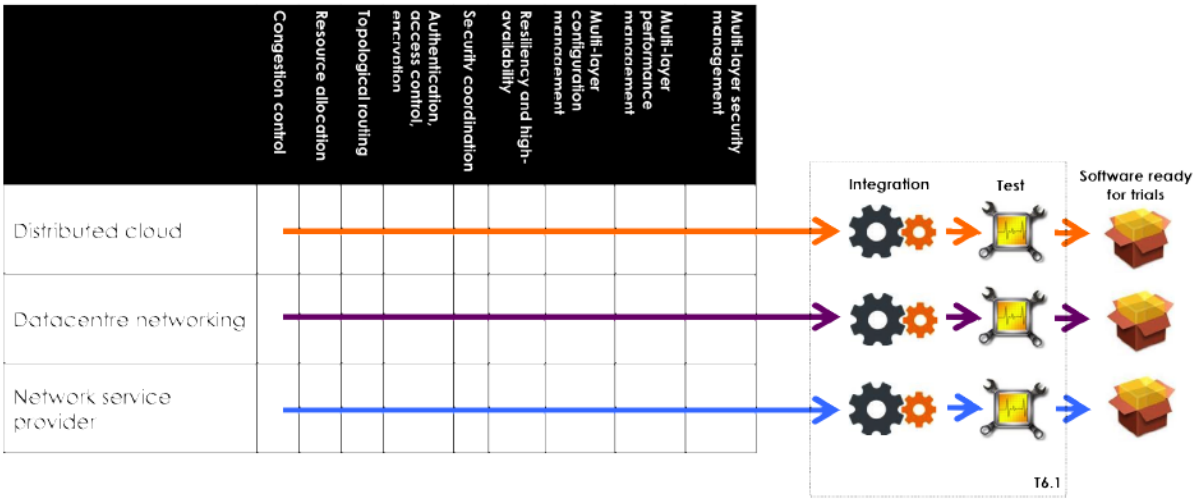

Figure 17. Illustration of WP6's strategy for system-level integration

As some of the software implementations are not still finished, and some of them are intended for simulation in the first phase of the project and implementation in the second one, for this first deliverable of WP6 we will collect the implementation contributions from a high-level point of view. This involves the implementation design, RINA policies that it addresses and the RINA components that are involved in each contribution. This has proven an effective approach, since this analysis has allowed us to provide valuable feedback to the respective WPs and partners with respect critical points from the integration perspective.

2.1. Parts to be integrated

In the following we describe the separate software components provided by Work Packages 3, 4 and 5, which form the units that are integrated in WP6 and whose integration will form the software bundles for each of the use cases. Some parts are intended for an specific use case, but here we collect all the different contributions without distinguishing their application in the use cases. In the next section, we put them within each use case and analyse the integration of the different parts involved.

2.1.1. Policies for resilient routing

Strategies for resilient routing involve primarily three PRISTINE stack components:

1. Routing module, that is in charge of computing the routing table from the DIF connectivity graph and exchanging (by means of the RIB daemon) link state advertisement with the other IPC Processes in the DIF
2. Resource allocator, that is in charge of computing the PDU forwarding table from the information contained in the routing table and push the resulting PDU forwarding table entires down to the kernel RMT component
3. RMT, the datapath element which chooses how to forward each ingress or egress PDU

The following modifications are required to those component in order to support resilient routing:

- Link-state routing is empowered with a new "Resiliency Routing" module that is charge of adding further Next Hop IPC Processes to the routing table entries computed by the main routing algorithm
- Resource Allocator must be able to specify multiple port ids for each destination in the routing table, in such a way that when a port-id gets unusable because of a failing link, alternate port-ids can be used
- RMT must be able, for each destination IPC Process, to switch between the main port-id and the alternate ones as soon as a failure is detected in the main one.

The IPC Process Daemon is implemented in C++ following an Object-Oriented (OO) approach. All the pluggable subcomponents are accessed by the IPC Process core through abstract interfaces (abstract C++ classes). Within the Routing component, the (existing) IRoutingAlgorithm interface is used to hide the details of the routing table computation - default plugin using the Dijkstra algorithm. In order to properly support resilient routing algorithm, a new IResiliencyAlgorithm interface has been added to

abstract the operations performed by the Routing component to modify (extend) the routing table computed by the routing algorithm.

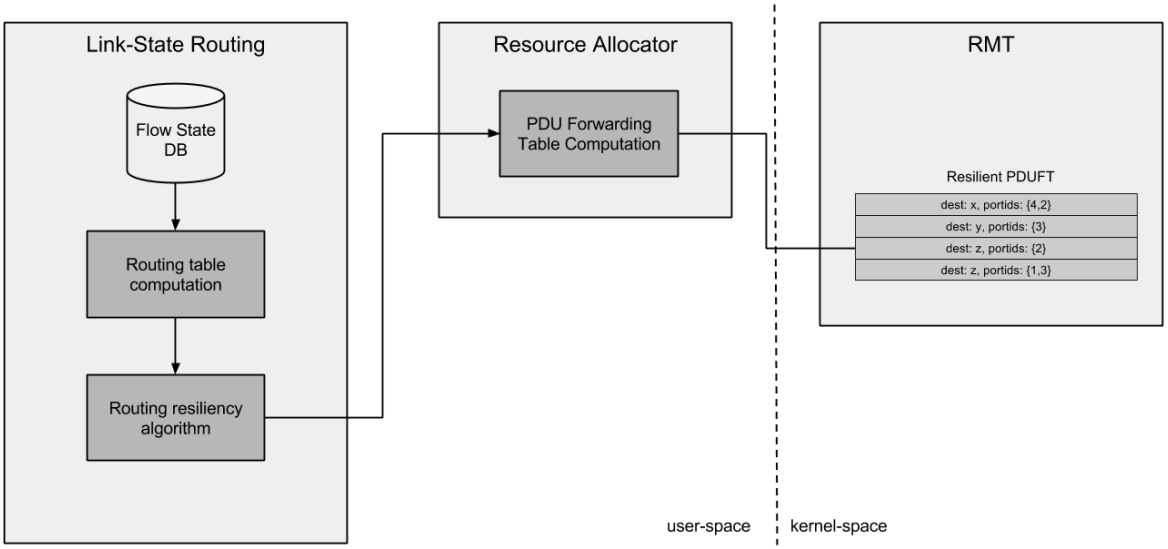

Figure 18. Resilient routing

2.1.2. Policies for QoS-aware Multipath routing

The QoS-aware Multipath Routing approach studied in PRISTINE is thoroughly described in [PRISTINE-D32]. It tackles the multipath routing strategies applied by a DIF based on the QoS requirements of the flow.

The next figure illustrates from a high-level point of view, the scope of this approach. The involved RINA components are the Relaying and Multiplexing Task (RMT), Resource Allocator (RA) and Routing.

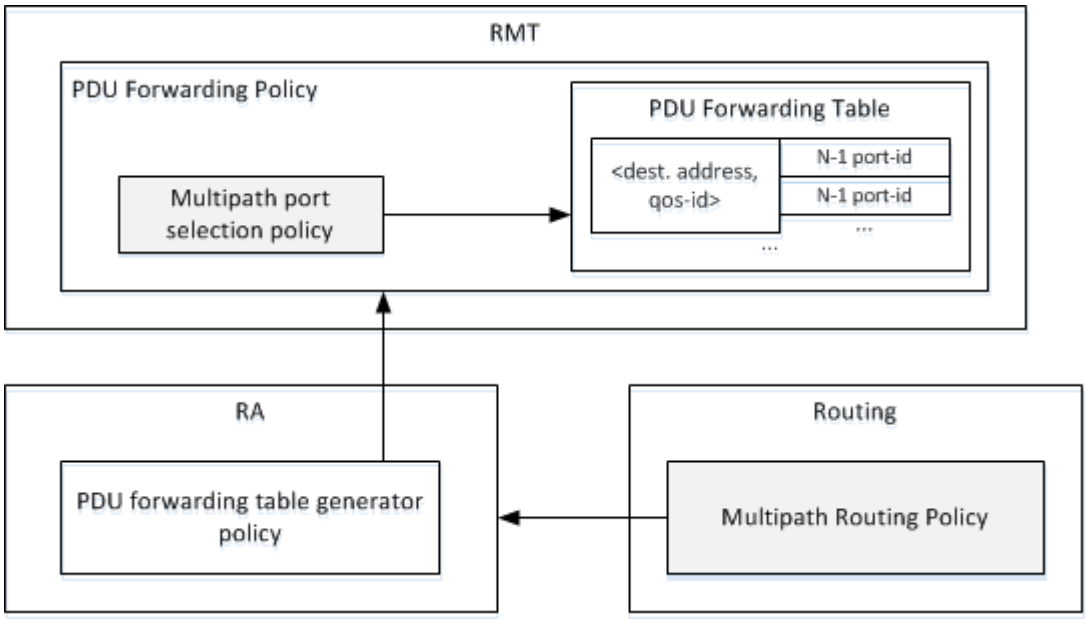

Figure 19. Multipath routing

- **Routing**: Routing computes routes to next hop addresses and provides this information to the Resource Allocator. Within Routing, the specific policy that applies for multipath is the so called multipath routing policy.
  - **Multipath routing policy**: This policy provides the necessary pre-computed information to optimize the forwarding decision time.
- **Resource Allocator**: The RA takes the output provided by the Routing component and provides information to the PDU Forwarding function in the RMT. The RA accounts for the adaptiveness of the multipath routing strategy, i.e. the dynamic modification of the forwarding table based on changing conditions. This is done by means of the dynamic update of the PDU forwarding table.
  - **PDU forwarding table generator policy**: The PDU forwarding table generator policy generates and updates the PDU forwarding table of the RMT. It takes as input information provided by the Resoruce Allocator, which includes aspects related to resource allocation policies.
- **Relaying and Multiplexing Task**: The primary job of the RMT in its multiplexing role is to pass PDUs from DTP instances to the appropriate (N-1)-ports. PDUs are usually forwarded based on its destination address, and QoS-cube-id fields. The key component of the RMT for multipath routing is the PDU forwarding policy.
  - **PDU forwarding policy**: This policy is invoked per PDU in order to obtain the N-1 port(s) through which the PDU has to be forwarded.
  - **PDU forwarding table**: The PDU forwarding policy consults the PDU forwarding table, which stores the pre-computed forwarding information.

2.1.3. Policies for Aggregate Congestion Control

Here, we specify the policies implemented/used for ACC in RINA.

RMT Policies

- **RMTQMonitorPolicy**: This policy can be invoked whenever a PDU is placed in a queue and may keep additional variables that may be of use to the decision process of the RMTSchedulingPolicy and RMTMaxQPolicy. Policy implementations:
  - REDMonitor: Using this implementation, if the average queue length is calculated and passed to ECNDropper.

- **RMTMAXQPolicy**: This policy is invoked when a queue reaches or crosses the threshold or maximum queue lengths allowed for this queue. Policy implementations:
  - ECNMarker: Using this implementation, if the queue length is exceeding its maximum threshold, RMT marks the current PDU by setting its ECN bit.
  - REDDropper: Using this implementation, if the queue length is exceeding its threshold, RMT drops the packet with a probability proportional to the amount exceeding the threshold.
  - UpstreamNotifier: By this implementation, if the queue size of RMT exceeds a maximum threshold when inserting a new PDU to the queue, RMT sends a direct congestion notification to the sender of that PDU using a CDAP message.
  - REDUpstreamNotifier: By this implementation, if the queue size of RMT exceeds a threshold when inserting a new PDU to the queue but it is shorter that the max threshold, RMT sends a direct congestion notification to the sender of that PDU using a CDAP message with a probability proportional to the amount exceeding the threshold.
  - ReadRateReducer: This implementation blocks the corresponding input queue of an output queue with a queue size exceeding its max threshold.
- **RMTSchedulingPolicy**: This is the core of RMT. This is the scheduling algorithm that determines the order in which input and output queues are served. This policy may be implemented by any of the standard scheduling algorithms, FCFS, LIFO, longestQfirst, priorities, etc.

EFCP Policies

- **DTCPTxControlPolicy**: This policy is used when there are conditions that warrant sending fewer PDUs than allowed by the sliding window flow control, e.g. the ECN bit is set in a control PDU. Policy implementations:
  - DTCPTxControlPolicyTCPTahoe: The pseudo code of this policy is shown in Algorithm 1. Procedure Initialize sets initial values of the local variables. In Send, if there is credit for transmission, it sends as many PDUs as allowed by the minimum of its send credit and the flow control window, and then, it adjusts its variables. If there is no credit remaining, it closes the DTCP window. In case of receiving acknowledgment packets, it adjusts cwnd and the other variables. If on any downstream link congestion happens, it receives a notification in the forms of direct CDAP messages, duplicate acknowledgment, or timeout. In the latter

case, it starts from slow start, and in the former cases, it halves its cwnd and goes to congestion avoidance.

- **DTPRTTEstimatorPolicy**: This policy is used to calculate RTT. Policy implementations:
  - DTPRTTEstimatorPolicyTCP: To calculate RTT and Retransmission Timeout (RTO) for DTCPTxControlPolicyTCPTahoe, this policy is run every time an acknowledgment is received. The pseudo code of this policy is shown in Algorithm 2.
- **DTCPSenderAckPolicy**: This policy is triggered when a sender receives an acknowledgment. Policy implementations:
  - DTCPSenderAckPolicyTCP: This policy performs the default bahavior of receiving ACKs, and counts the number of PDUs acknowledged. Then, it calls OnRcvACKs of DTCPTxControlPolicyTCPTahoe.
- **DTCPECNPolicy**: This policy is invoked upon receiving a PDU with ECN bit set in its header.
- **ECNSlowDownPolicy**: This policy is triggered when an explicit congestion notification is received from the RMT a congested relay node. The notification is sent as a CDAP message. Policy implementations:
  - TCPECNSlowDownPolicy: This policy only calls OnRcvSlowDown of DTCPTxControlPolicyTCPTahoe.

2.1.4. Policies for cherish/urgency multiplexing

Cherish/urgency scheduling is thoroughly described in [D32]. Cherish/urgency scheduling policies provide a way to distinguish flows with distinct QoS requirements of delay and drop probability, offering the resources of output ports in a prioritized way.

The following describes the policies necessary for cherish/urgency multiplexing, also referred as delay/loss scheduling. Moreover, policies for an enhanced version of the basic delay/loss scheduling are also specified. The next figure illustrates from a high-level point of view, the scope of this approach. The involved RINA components are the Relaying and Multiplexing Task (RMT) and Resource Allocator (RA).

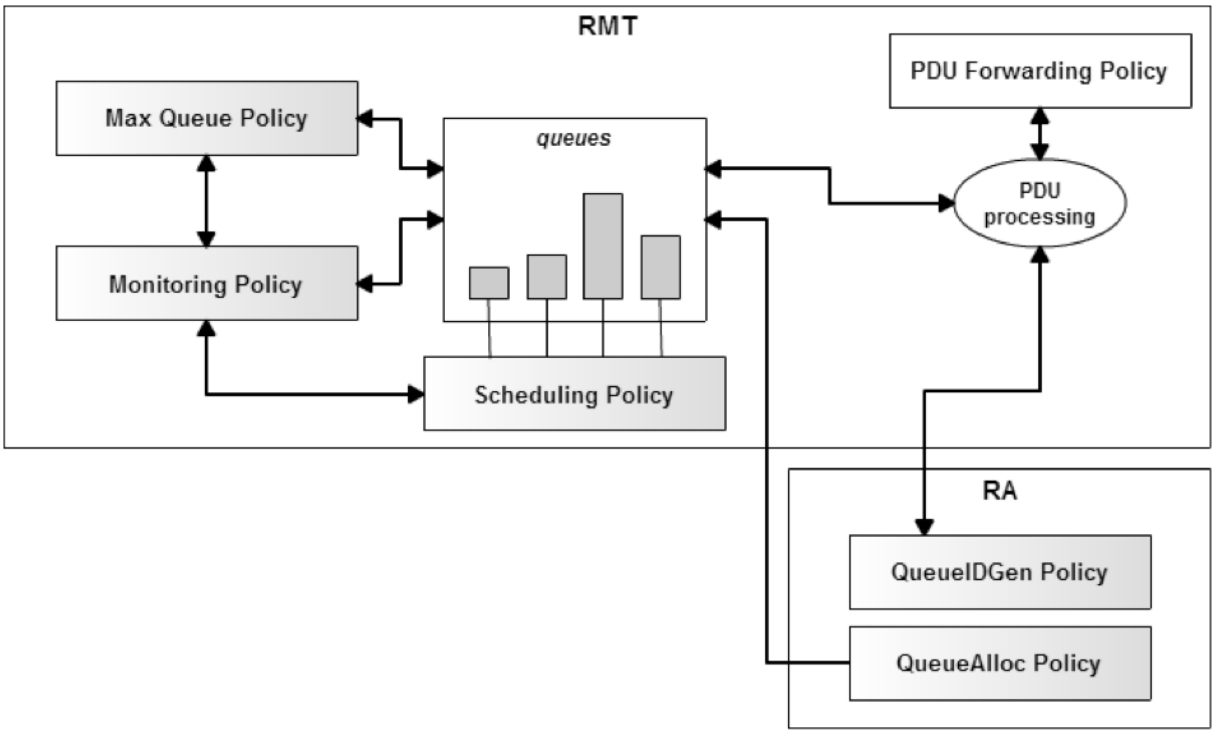

Figure 20. Cherish/Urgency scheduling related modules and policies

Resource Allocator

The RA has to create the distinct queues needed for delay/loss scheduling policies, as well as say the destination queue to incoming/outgoing PDUs.

- **QueueAlloc Policy**: This policy has to request the allocation of the required queues. In the delay/loss scheduling, we are going to create 1 queue at each input port and NxM queues at each output port, where N is the number of urgency levels in the DIF and M the number of cherish levels. In addition, management queues for CDAP messages are also created.
- **QueueIDGen Policy**: This policy has to map PDUs of a given port to a destination Queue. It requires a mapping of QoS class to Cherish/Urgency class and does the mapping to queues given the QoS Cube of the PDUs.

Relaying and Multiplexing Task

The primary job of the RMT in its multiplexing role is to pass PDUs from DTP instances and forwarding input PDUs from other IPCPs to the appropriate (N-1)-ports. When the appropriate (N-1)-port is selected, PDUs are placed into one of its queues based on a query to QueueIDGen policy.

- **Monitoring Policy**: This policy monitors the state of the distinct queues and computes data required for Max Queue and Scheduling policies. For delay/loss

and enhanced delay/loss scheduling policies, the monitoring policy is the core of scheduling. These policies have a common interface for the use of Max Queue and Scheduling policies returning the current state of a queue (occupation, threshold, drop probability, absolute threshold, as explained below) and the next queue to serve.

  - **Basic Delay/Loss Monitor**: This policy has a mapping "Queue to (UrgencyLevel + Threshold)", based on the Cherish/Urgency class assigned to the queue. It monitors the total amount of PDUs currently on all queues of an output port together, and when queried for the state of a queue, it returns the tuple <total occupation, threshold, 1, threshold>, indicating that a PDU has to be dropped if the "total occupation" of all queues of the port exceeds the "threshold" for that queue. In addition, it maintains a priority queue of queue pointers, used to respond to queries from the Scheduling policy for the next queue to serve. Queue pointers are inserted in this queue as PDUs arrive at the queues and use the queue urgency as priority value.
  - **Enhanced Delay/Loss Monitor**: This policy has a mapping "Queue to (UrgencyLevel + Threshold + DropProb + ThresholdAbsolute)", based on the Cherish/Urgency class assigned to the queue, and a mapping "UrgencyLevel to Skip Prob". As the simple Delay/Loss monitor, it monitors the total occupation per port. When queried for the state of a queue, it returns the tuple <total occupation, threshold, drop probability, threshold absolute>, indicating that a PDU has to be dropped if the "total occupation" of queues of the port related to the queue exceeds the "absolute threshold" for that queue, and if not, if it exceeds the "threshold" of the queue, it has to be dropped with "drop probability". It also maintains the same priority queue of queue pointers as in the basic version, but when queried for next queue to serve, it decides if an "urgency level/priority" is skipped given its "skip probability", resulting in a not strict order of priorities by urgency.

- **Max Queue Policy**: This policy has to decide if a PDU arriving to a queue has to be dropped or not. For Delay/Loss and enhanced Delay/Loss, this policy queries the monitor policy for the queue state (<occupation, threshold, dropProb, absThreshold>) and decides based on that:

```
if (occupation > absThreshold ) => Drop PDU
else if (occupation >  threshold and rand() < dropProb ) => Drop PDU
else => Accept PDU
```

- **Scheduling Policy**: This policy has to decide from which queue of a given port serve the next PDU when it is ready to serve. For Delay/Loss and enhanced Delay/Loss, this policy queries the monitor policy for the next queue to serve and simply serves that one.

(*) Management Queues are treated in a special way at monitoring policies, trying to avoid dropping PDUs, with the use of larger thresholds, and being always served first.

2.1.5. Policies for Congestion Control specific to the datacentre use case

These policies will be organized with a receiver detection, sender reaction approach. This means that the congestion is handled by marking the PDUs which passes through the network node as congested. At the PDU's destination EFCP instance there will be the logic necessary to compute some sort of reaction to mitigate the congestion effects. This logic will be based on the current state of the destination nodes and the receiving rate of the active EFCP instances on it. Finally a flow control PDU is emitted and received by the flow EFCP source instance, which reduce the transmission rate to the desired level.

Congestion control policies to integrate in the stack will require:

- **RMT policies**, monitors the incoming and outgoing packets in order to detect a congestion. If congestion takes place, then this policy decides which PDUs leave the IPCP with the congestion flag marked in its header. This policy can be seen as ECN marker policy with some logic to decide what queue has to be considered as congested. This decision obviously depend on how the policy decides to organize the queues. To obtain such behavior **Monitor** and **Scheduling** RMT policies will be introduced, in order to check the state of the queues and schedule the incoming/outgoing PDUs in the right order. The monitor policy will follows the queues load state, and will react to them once their load increases too much in too little time. The scheduling policy will move incoming PDUs in the right queues (depending on their organization) and will mark outgoing PDUs as congested in the queues that enter in a critical state.
- **EFCP transmission policies**, will be triggered by ECN-marked PDUs. When a PDU with the congested bit present in the header arrives, the policy becomes aware that the congestion is taking place or is going to be taking place soon in the route of the PDU. After this detection phase, the policy will compute the necessary adjustments in order to send to the connection's source EFCP instance

the new transmission rate, using the existing EFCP rate reduction mechanisms. To obtain such behavior **Transmission Control**, **Initial Rate** and **Rate Reduction** policies will be introduced. The Transmission control will be in charge of detecting congested marked packet an react to them. The Initial Rate policy will decide the initial rate which the flow will use to send the first PDUs. This depends on the actual rates used by the other EFCP instances, so a new component of the stack must be introduced, as a part of the Resource Allocator in the kernel side, to retrieve the necessary information. The Rate Reduction policy then will react to incoming feedback offered by the existing DTCP mechanisms, and will tune the DTCP instance to the ordered rate.

The following modifications are required to the following components in order to support congestion control:

- EFCP policy need to compute the transmission rate to send to the connection's source EFCP instance, and to do so it must obtain information over the status of the other EFCP instances currently active in the IPC process. The delegated component to support such information exchange is the Resource Allocator, which is located at the user-space area. Exchanging this type of information using the existing mechanism will greatly decrease the performances, because they have to cross twice the user/kernel barrier. A **kernel-level component** (part of the Resource Allocation domain) is needed in order to extract information of existing EFCP instances.

2.1.6. Policies for authentication and SDU Protection

Authentication

Authentication policies reside in the Security Manager component in user-space, and are used by the Enrollment Task policy set during the enrollment procedure to authenticate the remote IPC Process. As part of the authentication procedure, some data that may be required to successfully setup a potential encryption policy on that flow may be negotiated/generated, such as: the cipher to be used, its version and any parameters to configure them; the key to be used for encryption. If this is the case, the authentication policy stores this security context data in the Security Manager, which will use it to properly configure the SDU protection module in the kernel. The following authentication policies have been implemented:

- **No authentication**. A trivial policy for DIFs that do not require any authentication mechanism.

- **Password authentication**. As described in D4.2, the authenticator sends a random string challenge to the authenticating party, who performs an XOR operation with the random string and the password. The authenticating party returns the resulting string to the authenticator, who applies the same XOR operation and verifies that the recovered string is the same as the generated random string.
- **Cryptographic authentication(RSA)** (implementation in progress). The mechanics are similar to the password authentication, but Public Key Infrastructure is leveraged to perform the required encryption/decryption operation (as described in D4.2). Upon success, both parties generate a session key to be used for encryption.

SDU Protection

The SDU Protection module is a policy-based module that performs protection on outgoing SDUs and validates incoming SDUs before they are further processed. It is used by the RMT component. The following Figure shows where SDU protection resides in the IRATI RINA implementation, as well as its relationship with other relevant components.

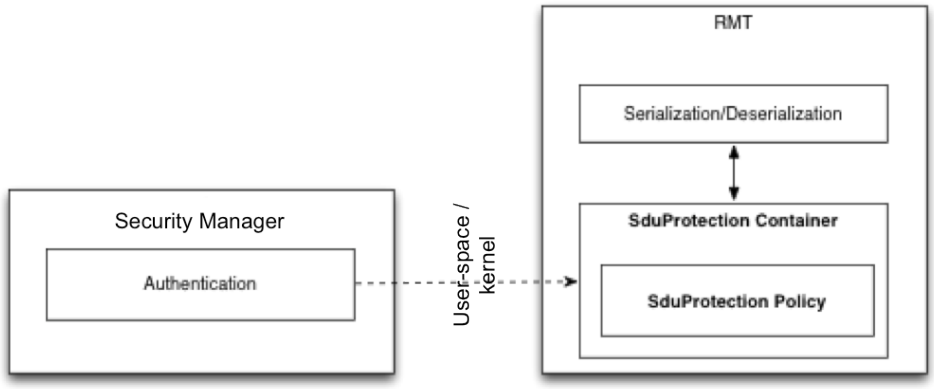

Figure 21. SDU Integration Block Diagram

The main components touched by the implementation of SDU Protection policy are:

- **RMT** component encapuslates functions related to SDU protection. RMT handles incoming and outging PDUs and forward them to the appropriate targets either locally or through external ports.
- **Serialization/Deserialization** is one of the subcomponents of RMT. It is called to serialize PDU to SDU before it can be sent to N-1 IPCP. On the other side, is

performs deserialization of incoming SDU to PDU before the PDU can be processed by the logics of RMT. Serialization/Deserialization is thus the source and sink of data for SDU protection.

- **SDU Protection** is a provided as a subcomponent of RMT. SDU protection performs its operations on serialized PDUs. It takes serialized PDUs and produces SDU. An SDU can contain additional information that are relevant to SDU protection methods applied, e.g., message authentication, SDU counter value, etc. SDU protection policy behaves according to settings provided by authentication module of EFCP.

To accomodate SDU policy modules, SDU protection provides a container that enables to install required policy, configure it and utilize it for performing SDU protection and verification operations. The following policies are available for SDU Protection:

- Simple CRC and TTL (SDUP-CRC-TTL) - This policy computes or checks the CRC on the SDU using specified CRC polynomial. It also computes and checks TTL.
- Cryptographic SDU Protection Policy based on AES Counter Mode (SDUP-CRYPTO-AES-CTR) - This policy protects SDUs by using cryptographic algorithms to prevent eavesdropping and tampering. Because of the way the SDU Protection Policy processes data, the counter-mode is only supported. In this policy AES algorithm is provided in two lengths, either 128 or 256. This is similar to AES utilization in TLS [tls]. For message integrity MD5 or SHA1 in different key lengths can be selected.

More SDU Protection Policies can be defined following the design of the provided policies.

2.1.7. Multi-Level Security (MLS)

Multi-Level Security (MLS) refers the protection of data or “objects” that can be classified at various sensitivity levels, from processes or “subjects” who may be cleared at various trusted levels. In [D4.1], we proposed a number of MLS architectures that enable secure data sharing to be achieved on the common RINA infrastructure. There are two components that are needed to create these MLS architectures:

- **Communications Security**, which protects the end-to-end transfer of data between IPC/application processes. This is needed to ensure that data cannot be inappropriately read from the communication channel and that data at different classification levels is not inappropriately mixed.

- **Boundary Protection Components** (BPC), which provide assured data flow between networks of differing sensitivity. They enable Low classified data residing on a High classified system to be moved to another Low classified system, while preventing accidental or deliberate release of sensitive information.

Communications Security

Options for implementing Communications Security in a RINA network are described in [D4.2]. The solution that could be produced to high levels of assurance is to implement communications security in dedicated devices that sit between the end device and an untrusted network, i.e. a network at a lower classification level. These devices encrypt PDUs before they are sent over a flow through a DIF that is at a lower classification level than the DIF it was sent from. Implementing such a device in RINA requires a customised SDU Protection Policy that applies protection to SDUs according to the classification level of the PDU and the classification level of the flow over which the SDU will be sent.

The next figure illustrates how the custom MLS Encryption Policy fits within RINA. The RINA components involved are the SDU Protection Module, RMT and the Authentication Module.

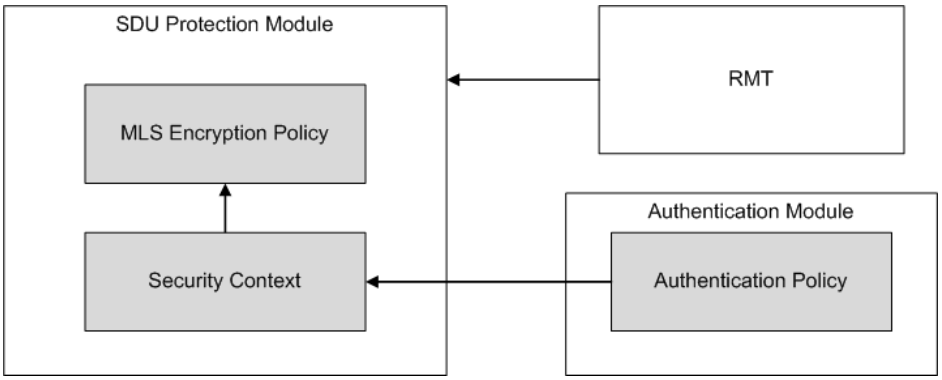

Figure 22. Block diagram of how MLS encryption policy fits in RINA

- **RMT:** this passes PDUs from DTP instances to the appropriate (N-1)-ports. Its serialisation task invokes the SDU Protection Module
- **Authentication Module:** during the enrolment process, this module authenticates the IPC process joining the DIF to ensure it is authorised.
  - **Authentication Policy:** this defines the authentication mechanism used to authenticate the joining IPCP. It also updates the SDU Protection Module's Security Context with any security parameters, e.g. key material and cryptographic algorithms, that may be negotiated as part of the authentication process

- **SDU Protection Module:** this applies protection to outgoing PDUs according to its policy
  - **MLS Encryption Policy:** this is an SDU Protection Policy that applies encryption to outgoing PDUs that are to be sent over a flow at a lower classification level.
  - **Security Context:** Contains the configuration data and security parameters needed by the SDU Protection policy, e.g. the encryption key and encryption algorithm to apply.

Boundary Protection Component

[D4.2] discusses options for implementing a BPC in RINA. The solution that could be produced to high levels of assurance is to implement the BPC at the DIF level, so that it is transparent to applications and cannot be bypassed. The BPC intercepts all PDUs and checks that they are appropriate for the destination before forwarding them.

The next figure illustrates how the BPC Policy fits within RINA. The RINA components involved are the underlying IPCP, RMT and Security Coordination.

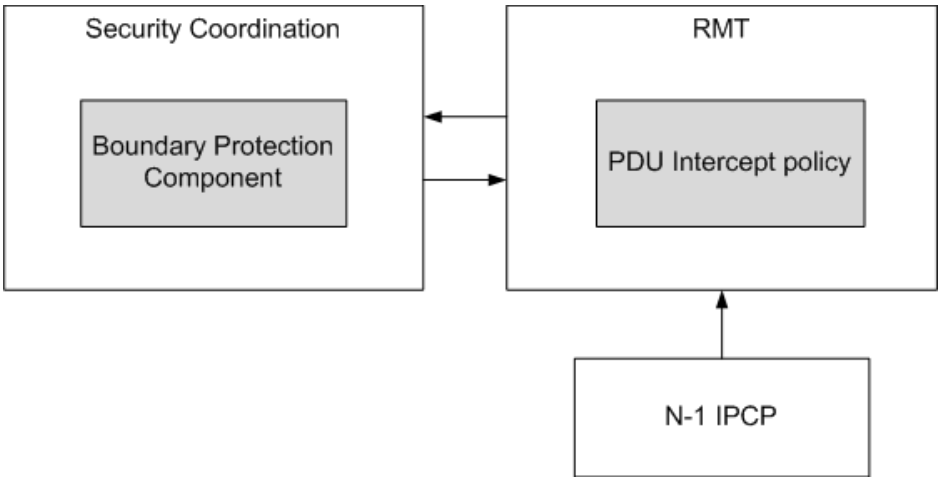

Figure 23. Block diagram of how BPC fits in RINA

- **N-1 IPCP:** this is the IPCP in the underlying DIF that passes the received PDU up to the N-level IPCP
- **RMT:** this receives PDUs from the N-1 flow and relays the PDU to either another N-1 flow or to the DTP instance depending on the destination address of the PDU.
  - **PDU Intercept Policy:** this policy intercepts all PDUs received by the IPCP, regardless of the intended destination. It passes the intercepted PDUs to Security Coordination to be inspected

- **Security Coordination:** this component is responsible for implementing a consistent security profile for the IPC Processes, coordinating the security-related functions .
  - **Boundary Protection Component:** this policy receives the intercepted PDUs and inspects them to ensure that they are appropriate for the destination, i.e. that they do not contain sensitive information. If the data is appropriate, it is allowed to be forwarded as normal; otherwise, the PDU is blocked.

2.1.8. Manager and Management Agent

More detail on the design of the Manager and Management Agent can be found in [D52]. The integration plan for the RINA stack, manager and management agent is a rather simple linear integration. The following integration sequence is appropriate:

1. RINA stack + DIF default policies.
2. Management Agent + RINA stack + DIF default policies
3. Manager + Strategies
4. Manager + Strategies + Management Agent + RINA stack + DIF default policies

The following sections detail some of the requirements on the defualt policy set and required management strategies.

Default policy set

For the most part the deafutl set of policies is sufficent to support management traffic over the NMS-DAF. However, a least tow types of QoSCubes will need to be supported:

a. CDAP commands, where the flow is bidirectional (between Manager and Management Agent), ie. responses are expected for most commands, therefore the same QoS Cube applies in both directions. Key requirements are: reliable retransmission (EFCP policy), with a guarenteed bandwidth (Resource allocation, DTP policies) and disconnect notifications (DTCP policy)

b. CDAP notifications, where the flow is unidirectional (from Management Agent to Manager). Key requirements are: reliable retransmission (EFCP policy), with a looser set of bandwidth, latency and jitter parameters (Resource allocation: QoS policies), and ideally whatevercast routing support (Addressing, NamespaceManagement policies).

The CDAP command QoS cube will also be used for RIB synchronisation, for example, new neighbour announcements.

Management strategies

The following management strategies are needed, which correspond to generic use-cases for the DMS system.

1. *shimipcp.CreationStrategy*. This strategy is unique as the shim IPCP provides no QoS classes, multiplexing support or EFCP mechanisms. A shim-IPC is a thin IPC layer over the existing VLAN or Ethernet functionalities.
2. *normalipcp.CreationStrategy*. This strategy is used to support two general types of flows, outlined above. In general, these are referred to a "CDAP command flow" and a "CDAP notification flow".
3. *normalipcp.DestructionStrategy*. This strategy is used to destroy normal-IPC Processes. However, from the manager viewpoint the CDAP commands are equivalent for shim-IPC Process destruction.
4. *managementagent.MonitoringStrategy*. This strategy is a delegatory strategy where the CDAP actions are issued by the sub-strategies. For example, to monitor a threshold value several sub-strategies may be employed depending on the exact context:
    a. An EventForwardingDiscriminator may need to be created
    $\Rightarrow$ *maEFD.CreationStrategy* (CDAP READ (EFDs), CDAP CREATE (EFD))
    b. Or an existing EventForwardingDiscriminator adjusted
    $\Rightarrow$ *maEFD.AdjustmentStrategy*, (CDAP READ (EFD old filter), CDAP STOP (EFD notifications), CDAP WRITE (new filter), CDAP START (EFD notifications))
    c. Or removed (as per adjusted unless the filter expression becomes empty)
    $\Rightarrow$ *maEFD.DestructionStrategy* (CDAP STOP (EFD), CDAP DELETE (EFD))

The above strategies give an insight into the type of management activities [D52] necessary for DMS operation. For example, performance and security monitoring can be seen as specialisations of a more generic monitoring strategy. Details on the operation of these strategies can be found in [D53].

2.2. Integration analysis

The goal of this section is to analyze, on a use case by use case basis, the integration of the various policies required in each one of the DIFs of the use case scenario, as well as to understand the implications for the Manager and Management Agent. The first

iteration of PRISTINE has focused on the development of policies to fulfill the needs of the Distributed Cloud and Datacentre Networking use cases, experimentation with the Network Service Provider use case has been left for the secont iteration of the project. As show in the Figure below, each experimental scenario consists on a series of computing systems running the IRATI stack with a number of policies (both at the kernel and user spaces), managed by an instance of the Network Manager via the Management Agent.

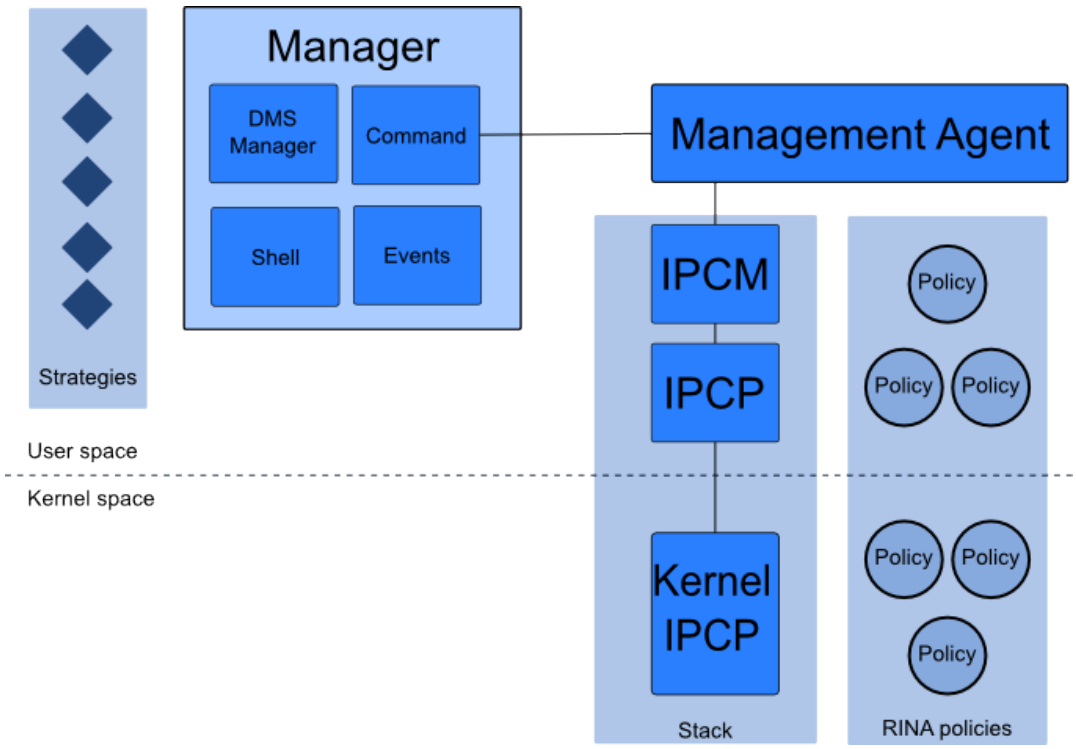

Figure 24. High level overview of the integration

2.2.1. Distributed cloud

Section 1.2.1 introduced the requirements of the different DIFs used in the distributed cloud use case. This section discusses the policies in each type of DIF that are required to fulfill those requirements, identifying any potential integration issues.

Tenant DIFs

These are very simple DIFs that support an application or a group of applications of a tenant. Tenant DIFs don't require routing policies or congestion control policies because all IPCPs are one hop away from each other. A simple FIFO scheduling policy is enough because the N-1 DIF (the cloud DIF) can provide flows with different quality of service. The only non-default policies that these type of DIFs may be using are:

- **Authentication**.

- No authentication, Password authentication or Cryptographic authentication. Depending on the strength of the authentication requirements, as well as on the need to support cryptographic encryption policies.

- **SDU Protection**.
  - Cryptographic encryption. May be enabled in some or all the N-1 flows, requires cryptographic authentication to generate a session key for encryption.

No policy integration conflicts are envisaged for tenant DIFs.

Cloud DIF

The cloud DIF connects all the VIFIB systems that can instantiate applications on behalf of users. It has to support differential traffic treatment via different classes of service, operate securely over public IP networks, and scale to a relatively large size. The cloud DIF supports the following policies.

- **Authentication**
  - Password authentication. For N-1 flows over Ethernet or over the backbone DIF.
  - Cryptographic authentication. For N-1 flows over public IP networks.
- **SDU Protection**
  - Cryptographic encryption. For N-1 flows over public IP networks.
- **Routing**
  - Link-state routing policy within each sub-DIF (region). No need of routing between regions since all border routers are one hope away.
- **Resource Allocator**
  - Upstream Notifier policy. To notify the resource allocators of IPCPs that host the EFCP instances that need to reduce their sending rates.
  - PDU Forwarding policy. Take the input of link-state routing (for intra-region routing) and inter-region N-1 flows to generate the forwarding table.
- **Relaying and Multiplexing Task**. The policies for the RMT have to provide traffic differentiation and monitoring/detection of congestion. Therefore the RMT policies for congestion control and for delay/loss multiplexing have to be properly integrated.
  - Monitor policy. The monitor policy of loss/delay multiplexing.

- Max queue policy. The monitor policy of the loss/delay multiplexing can be replaced by ECNMarker or UpstreamNotifier (to signal congestion and make the EFCP instances slow-down instead of directly dropping the PDUs at the RMT).
- Scheduling policy. The delay/loss scheduling policy.
- Forwarding policy. Longest match on destination address.

- **Error and Flow Control Protocol**.
  - The set of congestion-control related policies explained in section 2.1.3.

Therefore, this initial analysis identifies the RMT monitoring and maximum queue policies as the only potential source for integration problems. However, delay-loss multiplexing policies and RMT congestion detection policies can be easily combined, by having the delay/loss multiplexing approach replace the behaviour of dropping PDUs when a queue is full with earlier detection of queue growing and marking packets with the ECN flag.

Backbone DIF

The backbone DIF operates over public IP networks, provides a single class of service and needs to maximize the reliability of its flows.

- **Authentication**
  - Cryptographic authentication. For all N-1 flows.
- **SDU Protection**
  - Cryptographic encryption. For all N-1 flows.
- **Routing**
  - Link-state routing policy with the resilient routing algorithm that computes multiple next hop addresses for a each destination address.
- **RMT**
  - Monitor policy. Any of the policies described in section 2.1.3, required to implement detection and reaction to congestion.
  - Max queue policy. Any of the policies described in section 2.1.3, required to implement detection and reaction to congestion.
  - Scheduling policy. A simple FIFO policy.
  - Forwarding policy. Maintains backup port-ids for each destination address, switching to the backup port if it detects the primary option is no longer active.

- **Error and Flow Control Protocol**.
  - The set of congestion-control related policies explained in section 2.1.3.

No policy integration conflicts are envisaged for the backbone DIF.

2.2.2. Datacenter Networking

Among the previous parts, the policies and management strategies that are intended for the DatNet use case are:

- Policies for QoS-aware Multipath Routing.
- Policies for Aggregate Congestion Control (ACC).
- Policies for Cherish/Urgency Multiplexing.
- Congestion control in datacenter-use-case.
- Manager and Management Agent.

The next figure depicts an overview of the policies to be integrated for the DatNet use case. In the figure, the red lines denote the policies or interfaces in which two or more partners shall synchronize for the final implementation.

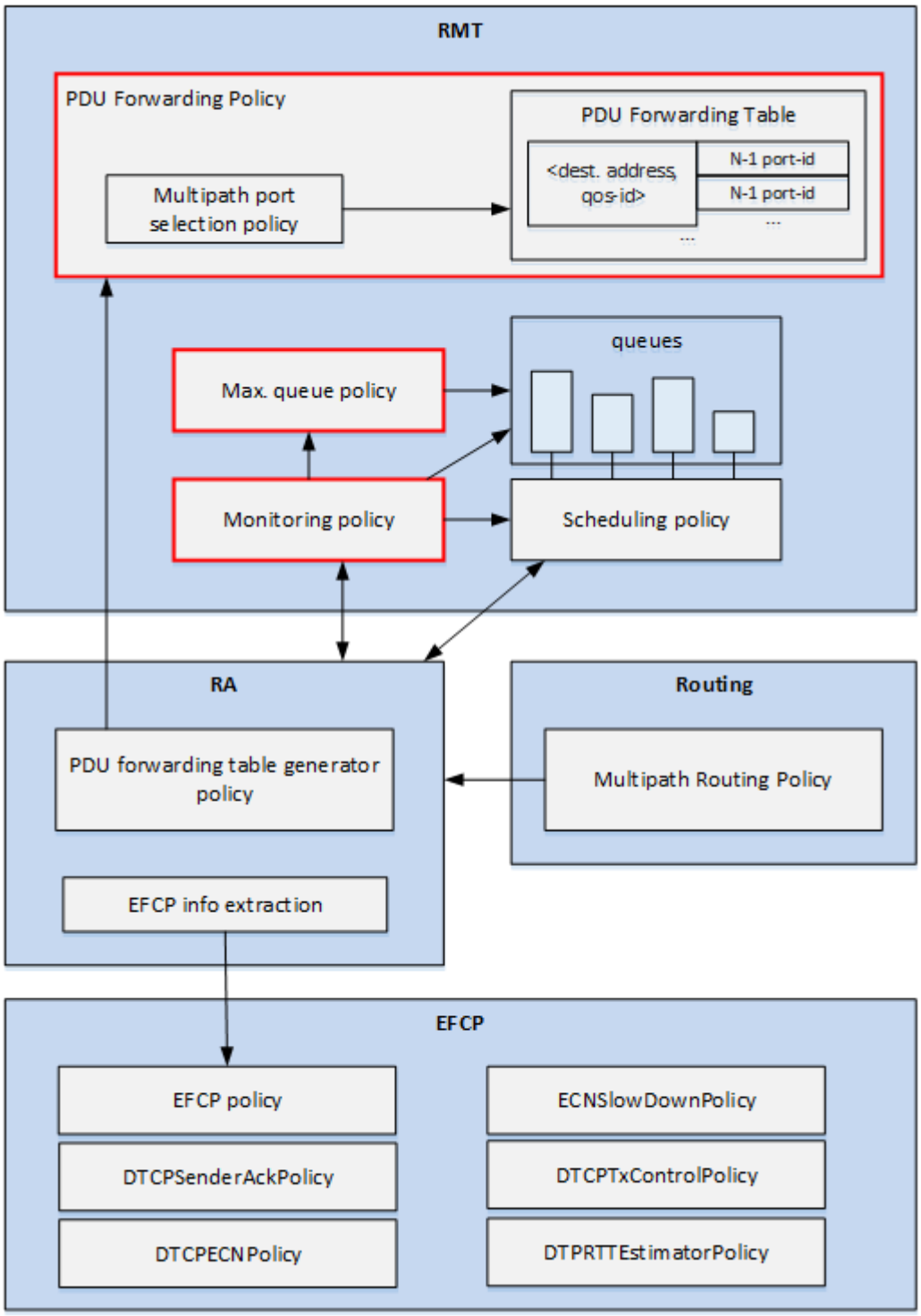

Figure 25. Overview of the policies for the DatNet use case

DatNet Policy Integration

The DIF structure of the DatNet use case consists on two DIFS: The Tenant DIF and the DC Fabric DIF. The requirements of the DIFs used in the distributed cloud use case were described in Section 1.2.2. This section discusses the distribution of policies among the datacenter DIFs' IPC processes, the policies in each type of DIF that are

required to fulfil the requirements identifying any potential integration issues and the global behaviour of the DIFs.

Tenant DIF

The Tenant DIF comprises the communications among VMs, servers and core routers. As in the DistCloud use case, these are simple DIF that support the communications of the virtual machines of a tenant. The policies that the Tenant DIF DIFs may be using are:

- **Routing**.
  - Simple routing policies are needed to forward the traffic among the Tenant DIF scope. Several core routers may be present, and the traffic shall be forwarded to one of them for the datacentre outgoing traffic. This decision can be made at the Tenant DIF. In the same way, for incoming traffic it shall be routed from the core routers to the corresponding server.
- **RMT**.
  - A simple FIFO scheduling policy is enough because the N-1 DIF (the DC Fabric DIF) can provide flows with different quality of service within the datacentre network.

No policy integration conflicts are envisaged for tenant DIFs.

DC Fabric DIF

The DC Fabric DIF connects all the nodes of the datacentre network, that is: servers, edge routers (or top of rack routers), aggregation routers and core routers. It supports differential traffic treatment via different classes of service. The DC Fabric DIF supports the following policies.

- **Routing**.
  - QoS-aware multipath routing. As the datacentre network is inherently a multipath topology (the topology design establishes different physical paths among the network nodes), the multipath routing strategies are a key aspect for the network performance. To that end, the multipath routing strategies described in section 2.1.2 will deal with the traffic forwarding among the different possible paths based on the QoS requirements aiming at achieving an optimal performance of the network.
- **Resource Allocator**.

- PDU forwarding table generator policy. Taking as input Routing and other information, it generates the PDU forwarding table of the PDU forwarding policy of the RMT.

- **Relaying and Multiplexing Task**. The policies for the RMT have to provide traffic differentiation and monitoring/detection of congestion. Therefore the RMT policies for congestion control and for delay/loss multiplexing have to be properly integrated.
  - Forwarding policy. It forwards packets through the corresponding ports based on the packet destination, QoS requirements and PDU forwarding table.
  - Monitor policy. Any of the policies described in section 2.1.3, required to implement detection and reaction to congestion.
  - Max queue policy. Any of the policies described in section 2.1.3, required to implement detection and reaction to congestion.
  - Scheduling policy. It provides traffic differentiation in terms of delay/loss scheduling.
- **Error and Flow Control Protocol**.
  - The set of congestion-control related policies explained in section 2.1.3.

Within the DC Fabric DIF, the initial analysis for the integration of the policies identifies the following potential points for integration conflicts:

- Monitor policy (RMT) and Max. queue policy (RMT). Problems may arise when integrating the delay-loss multiplexing policies and RMT congestion detection policies. However, they can be easily combined, by having the delay/loss multiplexing approach replace the behaviour of dropping PDUs when a queue is full with earlier detection of queue growing and marking packets with the ECN flag.

3. Integration Test plans

This section describes the strategy that will be followed to verify and validate that the RINA stack meets its design specifications.

The key aspects in which the test plans shall be focused are:

- Design Verification - to verify that the independent modules carry out the specified RINA functionality.
- Integration verification - to verify that the integrated system carries out the specified RINA functionality and that the modules' dependencies are correctly fulfilled.
- Performance tests - Once the system is correctly integrated and verified, performance tests shall be carried out to assess the system's performance according to the different test cases.

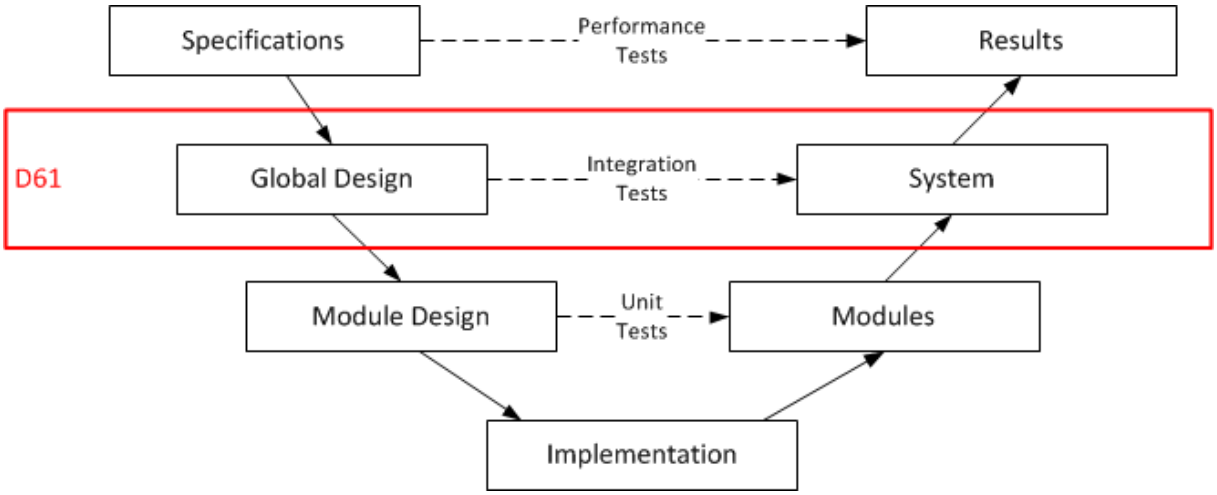

Figure 26. Testing methodology

3.1. Integration test case template

Table 1. Test case

Test Case:	{tc_number}	System:	{tc_system}
Test Case Name:	{tc_name}	Subsystem:	{tc_subsystem}
Short Description:	{tc_description}		
Configuration:	{tc_config}		

Steps

1. Do something

Expected System Response

1. Response d

Table 2. Results

Executed by:	{tc_by}		
Pass/Fail:	{tc_result}	Date:	{tc_date}
Comment:	{tc_comment}		

3.2. Common Integration Test Plans

In this section, first we describe the integration test plans that are common for all the use cases. These common test plans are grouped with regards the RINA stack and the management system. Next, in the following section, usecase-specific test plans are described for the DistCloud and DatNet use cases.

3.2.1. RINA stack

The goal of this validation activity is to detect any potential bugs that have been introduced as a result of the development of the Software Development Kit or due to structural changes required by specific policies. Specific policies related to the different use case scenarios have been validated at the system level via the specific test plans described later in this chapter. The following image shows the scenario used for the basic integration testing of the IRATI stack. The scenario has been designed to be simple but rich enough to verify the basic functionality of the RINA stack working with shim DIFs and a normal DIF configured with the IRATI stack default policies.

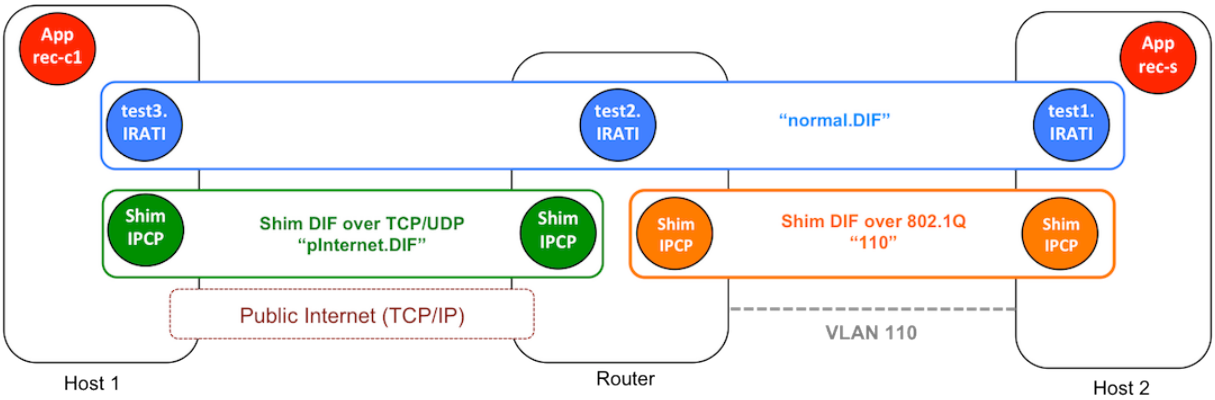

Figure 27. Scenario for the integration testing of the RINA stack with the default plugin

The RINA stack integration testing scenarios include three servers, with the following configuration (the complete configuration files are provided in Appendix A of this document):

- The first system (*Host1*) is configured with a shim IPC Process over TCP/UDP, a normal IPC Process and the *rina-echo* application running in *client* mode.
- The system in the middles (*Router*) is configured with two shim IPC Processes (one over Ethernet, the other over TCP/UDP), and a normal IPC Process.
- The last system (*Host2*) is configured with a shim IPC Process over Ethernet, a normal IPC process and the *rina-echo* application running in *server* mode.

The following subsections describe the different test cases used to validate the correct functioning of the RINA stack with the default policies, as well as the tests results obtained with the *pristine 1.2* branch of the *IRATI stack* github repository.

Test case 1: Creation of shim-eth-vlan IPC process

Test Case:	1	System:	RINA stack
Test Case Name:	Creation of shim-eth-vlan IPC Process	Subsystem:	Shim-eth-vlan IPC Process
Short Description:	Assuming the VLAN 110 is already configured in the interface *eth1* of host *Host 2*, this test will use the command line interface of the IPC Manager to create a shim IPCP of type *shim-eth-vlan*		
Configuration:	Host 2 configuration in Appendix A.1.3, without the *ipcProcessesToCreate* section		

Steps

- Execute the IPC Manager by typing the following instruction from the stack's installation path "/bin" folder.

```
./ipcm -c ../etc/ipcmanager.conf
```

- Telnet into the IPC Manager console by typing the following command.

```
telnet localhost 32766
```

- Create a *shim-eth-vlan* IPC Process by typing the following command.

```
IPCM >>> create-ipcp test-eth 1 shim-eth-vlan
```

```
IPC process created successfully [id = 1, name=test-eth]
```

Expected System Response

- Operation must be reported as successful. Querying the IPCPs via the IPC Manager console should provide the following output.

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
 | Port-ids of flows provided)
    1 | test-eth:1:: | shim-eth-vlan | INITIALIZED | - | -
```

Test case 2: Assignment of shim-eth-vlan IPC process to a DIF

Test Case:	2	System:	RINA stack
Test Case Name:	Assignment of shim-eth-vlan IPC Process to DIF	Subsystem:	Shim-eth-vlan IPC Process
Short Description:	Assuming test case 1 is already performed, this test will assign the shim IPCP created on test case 1 to the shim DIF *110*, allowing it to allocate flows and send data over the VLAN.		
Configuration:	Host 2 configuration in Appendix A.1.3, without the *ipcProcessesToCreate* section		

Steps

- Execute the following command in the IPC Manager console

```
IPCM >>> assign-to-dif 1 110
DIF assignment completed successfully
```

Expected System Response

- Operation must be reported as successful. Querying the IPCPs via the IPC Manager console should provide the following output.

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
 | Port-ids of flows provided)
    1 | test-eth:1:: | shim-eth-vlan | ASSIGNED TO DIF 110 | - | -
```

Test case 3: Creation of shim-tcp-udp IPC process

Test Case:	3	System:	RINA stack
Test Case Name:	Creation of shim-tcp-udp IPC Process	Subsystem:	Shim-tcp-udp IPC Process
Short Description:	Assuming the IP address 84.88.40.131 is already configured in one of the interfaces of *Host 2*, this test will use the command line interface of the IPC Manager to create a shim IPCP of type *shim-tcp-udp*		
Configuration:	Host 1 configuration in Appendix A.1.3, without the *ipcProcessesToCreate* section		

Steps

- Execute the IPC Manager by typing the following instruction from the stack's installation path "/bin" folder.

```
./ipcm -c ../etc/ipcmanager.conf
```

- Telnet into the IPC Manager console by typing the following command.

```
telnet localhost 32766
```

- Create a *shim-tcp-udp* IPC Process by typing the following command.

```
IPCM >>> create-ipcp test-tcp-udp 1 shim-tcp-udp
IPC process created successfully [id = 1, name=test-tcp-udp]
```

Expected System Response

- Operation must be reported as successful. Querying the IPCPs via the IPC Manager console should provide the following output.

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
 | Port-ids of flows provided)
    1 | test-tcp-udp:1:: | shim-tcp-udp | INITIALIZED | - | -
```

Test case 4: Assignment of shim-tcp-udp IPC process to a DIF

Test Case:	4	System:	RINA stack
Test Case Name:	Assignment of shim-tcp-udp IPC Process to DIF	Subsystem:	Shim-tcp-udp IPC Process
Short Description:	Assuming test case 3 is already performed, this test will assign the shim IPCP created on test case 3 to the shim DIF *pInternet.DIF*, allowing it to allocate flows and send data over the underlying IP layer (the public Internet in this case).		
Configuration:	Host 1 configuration in Appendix A.1.3, without the *ipcProcessesToCreate* section		

Steps

- Execute the following command in the IPC Manager console

```
IPCM >>> assign-to-dif 1 pInternet.DIF
DIF assignment completed successfully
```

Expected System Response

- Operation must be reported as successful. Querying the IPCPs via the IPC Manager console should provide the following output.

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
 | Port-ids of flows provided)
    1 | test-tcp-udp:1:: | shim-tcp-udp | ASSIGNED TO DIF pInternet.DIF |
 - | -
```

Test case 5: Creation of normal IPC process

Test Case:	5	System:	RINA stack
Test Case Name:	Creation of normal IPC Process	Subsystem:	Normal IPC Process
Short Description:	Assuming test case 2 has just been performed		
Configuration:	Host 2 configuration in Appendix A.1.3, without the *ipcProcessesToCreate* section		

Steps

- Create a *normal* IPC Process by typing the following command.

```
IPCM >>> create-ipcp test1.IRATI 1 normal-ipc
IPC process created successfully [id = 2, name=test1.IRATI]
```

Expected System Response

- Operation must be reported as successful. Querying the IPCPs via the IPC Manager console should provide the following output.

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
 | Port-ids of flows provided)
    1 | test-eth:1:: | shim-eth-vlan | ASSIGNED TO DIF 110 | - | -
    2 | test1.IRATI:1:: | normal-ipc | INITIALIZED | - | -
```

Test case 6: Registration of normal IPC process to N-1 DIF

Test Case:	6	System:	RINA stack
Test Case Name:	Registration of normal IPCP to N-1 DIF	Subsystem:	Normal and shim-eth-vlan IPCPs
Short Description:	Assuming test case 5 has just been performed, this test will register the *test1.IRATI:1* IPCP to the *110* shim DIF, allowing other IPCPs using the *110* shim DIF to allocate a flow to the *test1.IRATI:1* IPCP.		
Configuration:	Host 2 configuration in Appendix A.1.3, without the *ipcProcessesToCreate* section		

Steps

- Register the *normal* IPC Process to the N-1 DIF *110* by typing the following command.

```
IPCM >>> register-at-dif 2 110
IPC process registration completed successfully
```

Expected System Response

- Operation must be reported as successful. Querying the IPCPs via the IPC Manager console should provide the following output.

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
 | Port-ids of flows provided)
    1 | test-eth:1:: | shim-eth-vlan | ASSIGNED TO DIF 110 |
 test1.IRATI-1-- | -
    2 | test1.IRATI:1:: | normal-ipc | INITIALIZED | - | -
```

Test case 7: Assignment of normal IPC process to a DIF

Test Case:	7	System:	RINA stack
Test Case Name:	Assignment of normal IPC Process to DIF	Subsystem:	Normal IPC Process
Short Description:	Assuming test case 7 is already performed, this test will assign the IPCP created on test case 5 to the DIF *normal.DIF*		
Configuration:	Host 2 configuration in Appendix A.1.3, without the *ipcProcessesToCreate* section		

Steps

- Execute the following command in the IPC Manager console

```
IPCM >>> assign-to-dif 2 normal.DIF
DIF assignment completed successfully
```

Expected System Response

- Operation must be reported as successful. Querying the IPCPs via the IPC Manager console should provide the following output.

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
 | Port-ids of flows provided)
    1 | test-eth:1:: | shim-eth-vlan | ASSIGNED TO DIF 110 |
 test1.IRATI-1-- | -
```

```
2 | test1.IRATI:1:: | normal-ipc | ASSIGNED TO DIF normal.DIF | - | -
```

Test case 8: Destruction of IPC Processes

Test Case:	8	System:	RINA stack
Test Case Name:	Assignment of normal IPC Process to DIF	Subsystem:	Normal and shim IPCPs
Short Description:	Assuming test case 7 is already performed, this test will destroy both IPCPs and clean any related state.		
Configuration:	Host 2 configuration in Appendix A.1.3, without the *ipcProcessesToCreate* section		

Steps

- Execute the following command in the IPC Manager console

```
IPCM >>> destroy-ipcp 2
IPC process successfully destroyed
```

- Execute the following command in the IPC Manager console

```
IPCM >>> destroy-ipcp 1
IPC process successfully destroyed
```

Expected System Response

- Operation must be reported as successful. Querying the IPCPs via the IPC Manager console should provide the following output.

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
 | Port-ids of flows provided)
```

Test case 9: IPCP creation, registration to N-1 DIF and assignment to DIF via config file

Test Case:	9	System:	RINA stack

Test Case Name:	IPCP creation, registration to N-1 DIF and assignment to DIF via config file	Subsystem:	Normal and shim IPCPs
Short Description:	This test starts from scratch, and is performed in each of the three systems (*Host 1*, *Router*, *Host 2*). When the IPC Manager starts it will check the configuration file, create the required IPCPs, register them to one or more N-1 DIFs if needed and assign them to their respective DIFs.		
Configuration:	Host 1, Router and Host 2 configurations in Appendix A.1.3		

Steps (Repeat for each system)

- In system *Host 1*.Execute the IPC Manager by typing the following instruction from the stack's installation path "/bin" folder.

```
./ipcm -c ../etc/ipcmanager.conf
```

- Telnet into the IPC Manager console by typing the following command.

```
telnet localhost 32766
```

Expected System Response

- Operation must be reported as successful. Querying the IPCPs via the IPC Manager console should provide the following output.

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
 | Port-ids of flows provided)
    1 | test-tcp-udp:1:: | shim-tcp-udp | ASSIGNED TO DIF pInternet.DIF |
 test3.IRATI-1-- | -
    2 | test3.IRATI:1:: | normal-ipc | ASSIGNED TO DIF normal.DIF | - | -
```

Test case 10: Enrollment

Test Case:	10	System:	RINA stack

Test Case Name:	IPCP creation, registration to N-1 DIF and assignment to DIF via config file	Subsystem:	Normal and shim IPCPs
Short Description:	Assuming test 9 has been performed, this test will instruct the IPCP *test3.IRATI:1* to enroll with the neighbor IPCP *test2.IRATI:1* via the N-1 DIF *pInternet.DIF*, and also the IPCP *test1.IRATI:1* to enroll with the neighbor IPCP *test2.IRATI:1* via the N-1 DIF *110*.		
Configuration:	Host 1, Router and Host 2 configurations in Appendix A.1.3		

Steps (Repeat for each system)

- In system *Host 1*. Execute the following command from the IPC Manager console.

```
IPCM >>> enroll-to-dif 2 normal.DIF pInternet.DIF test2.IRATI 1
DIF enrollment succesfully completed
```

- In system *Host 3*. Execute the followign command from the IPC Manager console.

```
IPCM >>> enroll-to-dif 2 normal.DIF 100 test2.IRATI 1
DIF enrollment succesfully completed
```

Expected System Response

- Operation must be reported as successful. Querying the IPCPs via the IPC Manager console should provide the following output.

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
 | Port-ids of flows provided)
    1 | test-tcp-udp:1:: | shim-tcp-udp | ASSIGNED TO DIF pInternet.DIF |
 test3.IRATI-1-- | 1
    2 | test3.IRATI:1:: | normal-ipc | ASSIGNED TO DIF normal.DIF | - | -
```

Test case 11: Application registration

Test Case:	11	System:	RINA stack

Test Case Name:	Application registration	Subsystem:	Normal IPCP, test application
Short Description:	Assuming test 10 has been performed, this test will instruct the *rina-echo-time* application to register at the DIF *normal.DIF*		
Configuration:	Host 1 in Appendix A.1.3		

Steps

- In system *Host 1*. Execute the following command from the IRATI stack installation path, bin/ folder.

```
./rina-echo-time -l
5782(1429809221)#librina.logs (DBG): New log level: INFO
5782(1429809221)#librina.nl-manager (INFO): Netlink socket connected to
 local port 5782
```

Expected System Response

- Operation must be reported as successful. Querying the IPCPs via the IPC Manager console should provide the following output.

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
 | Port-ids of flows provided)
    1 | test-eth:1:: | shim-eth-vlan | ASSIGNED TO DIF 110 |
 test1.IRATI-1-- | 1
    2 | test1.IRATI:1:: | normal-ipc | ASSIGNED TO DIF normal.DIF | rina-
echo-server-1-- | -
```

Test case 12: Application flow allocation and deallocation

Test Case:	12	System:	RINA stack
Test Case Name:	Application registration	Subsystem:	Normal IPCP, test application
Short Description:	Assuming test 11 has been performed, this test will start the rina-echo-client application, which will allocate a flow to the server, exchange a number of PDUs and deallocate the flow.		

Configuration:	Host 2 in Appendix A.1.3

Steps

- In system *Host 3*. Execute the following command from the IRATI stack installation path, bin/ folder.

```
8154(1429810932)#librina.logs (DBG): New log level: INFO
8154(1429810932)#librina.nl-manager (INFO): Netlink socket connected to
 local port 8154
Flow allocation time = 44ms
SDU size = 20, seq = 0, RTT = 91ms
SDU size = 20, seq = 1, RTT = 103ms
SDU size = 20, seq = 2, RTT = 91ms
SDU size = 20, seq = 3, RTT = 103ms
SDU size = 20, seq = 4, RTT = 95ms
```

Expected System Response

- Operation must be reported as successful. Querying the IPCPs via the IPC Manager console should provide the following output.

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
 | Port-ids of flows provided)
    1 | test-tcp-udp:1:: | shim-tcp-udp | ASSIGNED TO DIF pInternet.DIF |
 test3.IRATI-1-- | 1
    2 | test3.IRATI:1:: | normal-ipc | ASSIGNED TO DIF normal.DIF | - | 2
```

3.2.2. Manager and Management Agent

The common validation of the Manager, Management Agent and RINA stack allows the verification of the correct operation of the Network Management System DAF (NMS-DAF). The figure below illustrates the scenario designed for this validation activity, building on the scenario of the RINA stack integration test. The goal is to perform the same actions of the previous validation activity (instantiation of IPC Processes, registration to N-1 DIFs, assignment to DIFs, enrollment) via the Management System. Therefore this scenario incorporates the entities that belong to the NMS-DAF (Management Agents and Manager), as well as a separate, dedicated DIF that supports the NMS-DAF (called NMS-DIF).

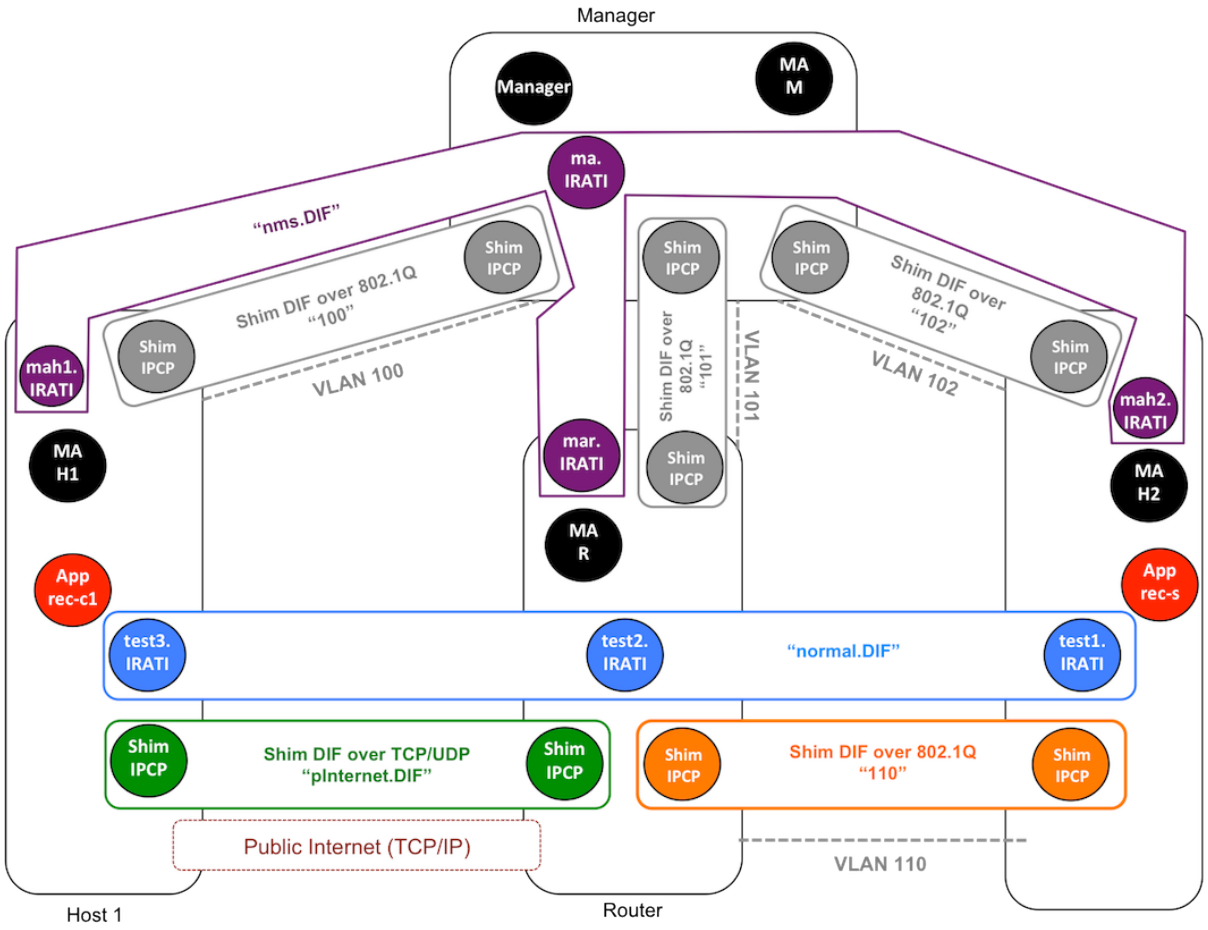

Figure 28. Scenario for the integration testing of the Manager, Management Agent and RINA Stack

The configuration of the three systems in section 4.2.1 is augmented with a shim IPCP to connect to the *Manager* system via a VLAN, and a normal IPCP that belongs to the *nms.DIF* DIF. Each system also has now a Management Agent, which communicates with the Manager process via the NMS DIF. The *Manager System* holds all the shim IPCPs required to communicate with the other three systems, the IPCP belonging to the *nms.DIF* DIF, the Management Agent and the Manager process. The following subsections describe the different test cases used to validate the correct functioning of the Manager, Management Agent and RINA stack with the default plugin.

Test plan 1: Bootstrap of Management Agent

Test Case:	1	System:	RINA stack, Management Agent, Manager
Test Case Name:	Bootstrap of Management	Subsystem:	Shim-eth-vlan IPC Process

	Agent: enrollment to NMS-DAF		
Short Description:	Register the Management agent to the NMS-DAF		
Configuration:	Host 1 configuration in Appendix A.2.3		

Steps

- Execute the IPC Manager by typing the following instruction from the stack's installation path "/bin" folder.

```
./ipcm -c ../etc/ipcmanager.conf
```

Expected System Response

- Operation must be reported as successful. Querying the IPCPs via the IPC Manager console should provide the following output.

```
IPCM >>> list-ipcps
Current IPC processes
(id | name | type | state | Registered applications | Port-ids of flows
 provided)
    1 | test-eth:1:: | shim-eth-vlan | INITIALIZED | - | -
```

Test plan 2: Bootstrap of Manager

It is advisibile to start the DMS manager first, as Management Agents attempt to connect to the DMS manager after NMS-DAF enrolment.

Test Case:	2	System:	RINA stack, Management Agent, Manager
Test Case Name:	Bootstrap of Manager: enrollment to NMS-DAF	Subsystem:	Shim-eth-vlan IPC Process
Short Description:	Register the Manager to the NMS-DAF		
Configuration:	Host 1 configuration in Appendix A.2.3		

Steps

- Execute the IPC Manager by typing the following instruction from the stack's installation path "/bin" folder.

```
./ipcm -c ../etc/ipcmanager.conf
```

- Execute the DMS Manager by typing the following instructions from the manager's installation path "/all-demos/dms/bin" folder.

```
./dms.sh
```

Expected System Response

- Four windows are created containing, an event logging console, a DMS command console, a DMS manager console, and the MA event simulation console. (On headless systems, four processes are started in the background)
- Startup must be reported as successful, in all consoles, with no stack traces visible.

Test plan 3: Manager self test

Test Case:	3	System:	RINA stack, Management Agent, Manager
Test Case Name:	Self test of Manager	Subsystem:	Manager
Short Description:	Verify communications within manager components, and simulation console.		
Configuration:	Host 1 configuration in Appendix A.2.3		

Steps

- Locate the DMS command console. Type:

```
run selftests
```

Expected System Response

- Some activity will be seen in the DMS manager console.
- A log message is generated indicating the emmission of the CDAP_READ event.

```
<timestamp> CDAP read (invoke_id=xx) (computing_system_id=1,
 processing_system_id=7, ipc_process_id=30,qos_cube_id=97)
```

Test plan 4:Instantiation of a shim IPC Process

Test Case:	4	System:	RINA stack, Management Agent, Manager
Test Case Name:	Creation of shim IPC Process	Subsystem:	All
Short Description:	Create a new shim IPC process in the attached Management Agent.		
Configuration:	Host 1 configuration in Appendix A.2.3		

TODO: the host configuraiton needs to be checked in Appendix A, does the shim ipc process already exist?

Steps

- The manual trigger targets a IPC process with the RDN (computing_system_id=1, processing_system_id=7, ipc_process_id=30)
- Locate the DMS command console. Register the required strategies and triggers by typing:

```
using eu.ict_pristine.wp5.dms.strategies
sm registerTriggerClass className:shimipcp.CreationTrigger
sm registerStrategyClass className:shimipcp.CreationStrategy
sm deployStrategy className:shimipcp.CreationStrategy, id:shimipcpcreate,
trigger:shimipcp.CreationTrigger
sm activateStrategy id:shimipcpcreate
```

- In the DMS command console. Manually trigger the shim IPC Process creation, by typing:

```
ts sendTrigger id:shimipcpcreate
```

Expected System Response

- Several log messages are generated indicating execution of the Strategies.

- The Manager event console should have the following events logged, where the invoke_ids correlate:

```
<timestamp> CDAP create (invoke_id=xx) (computing_system_id=1,
 processing_system_id=7, ipc_process_id=30)
... other events
<timestamp> CDAP create reply (invoke_id=xx, result=0)
 (computing_system_id=1, processing_system_id=7, ipc_process_id=30)
```

- The Management Agent the CDAP create operation must be reported as successful.
- In the Management Agent, querying the IPCPs via the IPC Manager console should provide include following output.

```
IPCM >>> list-ipcps
Current IPC processes
(id | name | type | state | Registered applications | Port-ids of flows
 provided)
1 | test-eth:1:: | shim-eth-vlan | INITIALIZED | - | -
```

Test plan 5: First normal IPC Process in a DIF

This test plan refers to the creation, assignment and registration of a single IPCP in a DIF. This is the first IPCP to be created in the DIF. This IPCP contains the DIF configuration so other IPCPs can join the DIF without the complete DIF specification.

Test Case:	5	System:	RINA stack, Management Agent, Manager
Test Case Name:	Creation of normal IPC Process	Subsystem:	All
Short Description:	Create a new normal IPC process in the attached Management Agent.		
Configuration:	Host 1 configuration in Appendix A.2.3		

Steps

- The manual trigger targets a IPC process with the RDN (computing_system_id=1, processing_system_id=7, ipc_process_id=47)
- Locate the DMS command console. Register the required strategies and triggers by typing:

```
using eu.ict_pristine.wp5.dms.strategies
sm registerTriggerClass className:normalipcp.CreationTrigger
sm registerStrategyClass className:normalipcp.CreationStrategy
sm deployStrategy   className:normalipcp.CreationStrategy,
id:normalipcpcreate, trigger:normalipcp.CreationTrigger
sm activateStrategy id:normalipcpcreate
```

- In the DMS command console. Manually trigger the normal IPC Process creation, by typing:

```
ts sendTrigger id:normalipcpcreate
```

Expected System Response

- Several log messages are generated indicating execution of the Strategies.
- The Manager event console should have the following events logged:

```
<timestamp> CDAP create (invoke_id=xx) (computing_system_id=1,
 processing_system_id=7, ipc_process_id=47)
... other events
<timestamp> CDAP create reply (invoke_id=xx, result=0)
 (computing_system_id=1, processing_system_id=7, ipc_process_id=47)
```

- The Management Agent the CDAP create operation must be reported as successful.
- In the Management Agent, querying the IPCPs via the IPC Manager console should provide include following output.

```
IPCM >>> list-ipcps
Current IPC processes
(id | name | type | state | Registered applications | Port-ids of flows
 provided)
1 | test-eth:1:: | shim-eth-vlan | INITIALIZED | - | -
2 | test-eth:1:: | normal-ipcp | INITIALIZED | - | -
```

Test plan 6: Destruction of an IPC Process

Test Case:	6	System:	RINA stack, Management Agent, Manager

Test Case Name:	Destruction of normal IPC Process	Subsystem:	All
Short Description:	Destruction of a normal IPC process in the attached Management Agent.		
Configuration:	Host 1 configuration in Appendix A.2.3		

Steps

- The manual trigger targets a IPC process with the RDN (computing_system_id=1, processing_system_id=7, ipc_process_id=47)
- Locate the DMS command console. Register the required destruction strategies and triggers by typing:

```
using eu.ict_pristine.wp5.dms.strategies
sm registerTriggerClass className:normalipcp.DestructionTrigger
sm registerStrategyClass className:normalipcp.DestructionStrategy
sm deployStrategy   className:normalipcp.DestructionStrategy,
id:ipcpdestroy, trigger:normalipcp.DestructionTrigger
sm activateStrategy id:ipcpdestroy
```

- In the DMS command console. Manually trigger the destruction of the normal IPC Process, by typing:

```
ts sendTrigger id:ipcpdestroy
```

Expected System Response

- Several log messages are generated indicating execution of the Strategies.
- The Manager event console should have the following events logged:

```
<timestamp> CDAP create (invoke_id=xx) (computing_system_id=1,
 processing_system_id=7, ipc_process_id=47)
... other events
<timestamp> CDAP create reply (invoke_id=xx, result=0)
 (computing_system_id=1, processing_system_id=7, ipc_process_id=47)
```

- The Management Agent the CDAP DELETE operation must be reported as successful.

- In the Management Agent, querying the IPCPs via the IPC Manager console should confirm the non-existance of the normal IPC Process.

```
IPCM >>> list-ipcps
Current IPC processes
(id | name | type | state | Registered applications | Port-ids of flows
 provided)
1 | test-eth:1:: | shim-eth-vlan | INITIALIZED | - | -
```

Test plan 7: Subsequent normal IPC processes

This test plan covers the creation and enrollment of an IPC Process to a neighbor that is **already** a member of the DIF. This verifies the way that an IPC Process without the complete DIF configuration can join an existing DIF.

Test Case:	7	System:	RINA stack, Management Agent, Manager
Test Case Name:	Subsequent normal IPC processes	Subsystem:	All
Short Description:	Create a new normal IPC process and enroll in an existing DIF.		
Configuration:	Host 1 configuration in Appendix A.2.3		

Steps

- The manual trigger targets a IPC process with the RDN (computing_system_id=1, processing_system_id=7, ipc_process_id=48)
- Locate the DMS command console. Register the required strategies and triggers by typing:

```
using eu.ict_pristine.wp5.dms.strategies
sm registerTriggerClass className:normalipcp.CreationTrigger
sm registerStrategyClass className:normalipcp.CreationStrategy
sm deployStrategy   className:normalipcp.CreationStrategy,
id:normalipcpcreate, trigger:normalipcp.CreationTrigger
sm activateStrategy id:normalipcpcreate
```

- In the DMS command console. Manually trigger the normal IPC Process creation, by typing:

```
ts sendTrigger id:normalipcpcreate
```

Expected System Response

- Several log messages are generated indicating execution of the Strategies.
- The Manager event console should have the following events logged:

```
<timestamp> CDAP create (invoke_id=xx) (computing_system_id=1,
 processing_system_id=7, ipc_process_id=47)
... other events
<timestamp> CDAP create reply (invoke_id=xx, result=0)
 (computing_system_id=1, processing_system_id=7, ipc_process_id=47)
```

- The Management Agent the CDAP create operation must be reported as successful.
- In the Management Agent, querying the IPCPs via the IPC Manager console should provide include following output.

```
IPCM >>> list-ipcps
Current IPC processes
(id | name | type | state | Registered applications | Port-ids of flows
 provided)
    1 | test-tcp-udp:1:: | shim-tcp-udp | ASSIGNED TO DIF pInternet.DIF |
 test3.IRATI-1-- | 1
    2 | test3.IRATI:1:: | normal-ipc | ASSIGNED TO DIF normal.DIF | - | -
```

Test plan 8: Subsequent normal IPC process: enrollment failure

This test plan covers the creation and enrollment of an IPC Process to a neighbor that is **already** a member of the DIF. This verifies the way that an IPC Process without the complete DIF configuration can attempt to join an existing DIF, but this enrolment fails, as the configuration is for a different DIF.

Test Case:	8	System:	RINA stack, Management Agent, Manager
Test Case Name:	Subsequent normal IPC process: enrollment failure	Subsystem:	All

Short Description:	Create a new normal IPC process, enroll in an existing DIF, the enrol failes.
Configuration:	Host 1 configuration in Appendix A.2.3

Steps

- The manual trigger targets a IPC process with the RDN (computing_system_id=1, processing_system_id=7, ipc_process_id=48)
- Locate the DMS command console. Register the required strategies and triggers by typing:

```
using eu.ict_pristine.wp5.dms.strategies
sm registerTriggerClass className:normalipcp.CreationTrigger
sm registerStrategyClass className:normalipcp.ErrCreationStrategy
sm deployStrategy   className:normalipcp.CreationStrategy,
id:normalipcperrcreate, trigger:normalipcp.CreationTrigger
sm activateStrategy id:normalipcperrcreate
```

- In the DMS command console. Manually trigger the normal IPC Process creation, by typing:

```
ts sendTrigger id:normalipcperrcreate
```

Expected System Response

- Several log messages are generated indicating execution of the Strategies.
- The Manager event console should have the following events logged:

```
<timestamp> CDAP create (invoke_id=xx) (computing_system_id=1,
 processing_system_id=7, ipc_process_id=48)
... other events
<timestamp> CDAP create reply (invoke_id=xx, result=0)
 (computing_system_id=1, processing_system_id=7, ipc_process_id=48)
```

- The Management Agent the CDAP create operation must be reported as successful.
- In the Management Agent, querying the IPCPs via the IPC Manager console should provide include following output.

```
IPCM >>> list-ipcps
```

```
Current IPC processes
(id | name | type | state | Registered applications | Port-ids of flows
 provided)
1 | test-tcp-udp:1:: | shim-tcp-udp | ASSIGNED TO DIF pInternet.DIF |
 test3.IRATI-1-- | 1
2 | test3.IRATI:1:: | normal-ipc | INITIALIZED | - | -
```

3.3. Specific test plans

Since each of the policies contributed to WP6 has been already verified by the work package that developed the policy, specific WP6 tests only need to target the validation of the policies whose integration can be problematic. Chapter 2 of this document has only identified a single set of potential conflictive policies for each of the two use cases:

- The delay-loss multiplexing policies and the RMT congestion detection policies for the datacentre networking and the distributed cloud use cases.

3.3.1. Test case 1: Behavior of delay/loss multiplexer under congestion conditions

Test Case:	1	System:	RINA stack, Management Agent, Manager, ACC policies, delay-loss multiplexing policies
Test Case Name:	Behavior of delay/ loss multiplexer under congestion conditions	Subsystem:	EFCP and RMT
Short Description:	The goal of this test is to verify the correct behavior of the delay-loss multiplexer working together with congestion-control related policies. In the absence of congestion management policies, when the queues of the delay-loss multiplexer would get full, the policy would discard any incoming PDUs, triggering retransmissions in the case the PDUs were part of reliable flows. If congestion management is introduced, the DIF can react when the queues of the delay-		

	loss multiplexer approach its maximum size (90% is the value used for this test). When the queues reach that threshold, the RMT will mark the PDUs with the ECN flag, causing the EFCP instances that process these PDUs at destination to decrease the rate of the sender, thus reducing the level of congestion in the DIF without the need of dropping PDUs. Experiments in T6.2 can try modifications of these basic behaviour, the goal of T6.1 is just to validate the correct operation of the basic mechanism.
Configuration:	Departing from the configuration shown in Figure 27, with the delay-loss multiplexing policies and aggregate congestion control policies as part of the DIF definition. Execute the tests two times: the first one without aggregate congestion control policies, the second one with aggregate congestion control policies. The test starts with the three IPCPs of the normal DIF already enrolled, ready to provide flows.

Steps:

- Use the Linux network emulation (*netem*) toolkit [netem] to limit the throughput of VLAN 110.

```
tc qdisc add dev eth1.110 parent 1:1 handle 10: tbf rate 128kbit buffer
 1600 limit 3000
```

- Execute the rina-echo-time application in "server mode", launching it in host 2.

```
./rina-echo-time -l
```

- Execute in parallel 4 rina-echo-time applications in host 1, each one requesting a reliable flow with a different qos requirements (low loss/low delay; high loss/low delay; etc)

```
(these are the contents of a script)
./rina-echo-time -w 0 -s 1400 -c 10000 -g 0 --client-api 1 -qos 0 &
./rina-echo-time -w 0 -s 1400 -c 10000 -g 0 --client-api 2 -qos 1 &
./rina-echo-time -w 0 -s 1400 -c 10000 -g 0 --client-api 3 -qos 2 &
./rina-echo-time -w 0 -s 1400 -c 10000 -g 0 --client-api 4 -qos 3 &
```

- Check the reported delay and PDU loss for each instance of the rina-echo-time application

Expected System Response

- In both executions, the flows belonging to the low-loss category should report similar values of PDU loss, lower than the values reported by the flows belonging to the high-loss category. The same should happen with the delay.
- In the second execution there should be less PDU loss (since congestion control policies were active), and also the values of the reported delay should be slightly lower due to the lower number of retransmissions.

4. Future plans

In this first deliverable of WP6 we have presented the first steps on the integration effort. This effort has involved a tight coordination between the different partners to define their implementation scope, to provide their inputs to WP6 and to identify potential conflict points between the different parts.

For the future PRISTINE's integration work, WP6 will address the integration of real implementations and their deployment over the Virtual Wall testbed, checking that the integration test plans are correctly fulfilled and providing feedback to the respective work packages with the integration test results and indications to solve integration issues that might appear.

Future work will also cover the trials and results analysis withing the scope of tasks T6.2 and T6.3.

List of definitions

Application Process (AP)

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

Common Application Connection Establishment Phase (CACEP)

CACEP allows Application Processes to establish an application connection. During the application connection establishment phase, the APs exchange naming information, optionally authenticate each other, and agree in the abstract and concrete syntaxes of CDAP to be used in the connection, as well as in the version of the RIB. It is also possible to use CACEP connection establishment with another protocol in the data transfer phase (for example, HTTP).

Common Distributed Application Protocol (CDAP)

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

Distributed Application Facility (DAF)

A collection of two or more cooperating APs in one or more processing systems, which exchange information using IPC and maintain shared state.

Distributed-IPC-Facility (DIF)

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

IPC-Process

An Application-Process, which is a member of a DIF and implement locally the functionality to support and manage IPC using multiple sub-tasks.

List of acronyms

AP	Application Process
CACEP	Common Application Connection Establishment Phase
CDAP	Common Distributed Application Protocol
DAF	Distributed Application Facility
DAP	Distributed Application Process
DIF	Distributed-IPC-Facility
DMS	Distributed Management System
DTCP	Data Transfer and Control Protocol
ECN	Explicit Congestion Notification
EFCP	Error and Flow Control Protocol
IPC	Inter-Process Communication
IPCP	Inter-Process Communication Process
IPCM	Inter-Process Communication Manager
LIFO	Last In - First Out
NM	Network Management
PDU	Protocol Data Unit
RA	Resource Allocation
RDN	Relative Distinguished Name
RIB	Resource Information Base
RINA	Recursive Inter-Network Architecture
RMT	Relaying an Multi-plexing Task
RTO	Retransmission Time Out
RTT	Round Trip Time
SDK	Software Development Kit
SDU	Service Data Unit
SFF	Service Function Forwarder
VLAN	Virtual LAN
VM	Virtual Machine

References

- [D23] PRISTINE Consortium. Deliverable D2.3. Proof of concept of the Software Development Kit. January 2015. Available online[1].
- [D32] PRISTINE Consortium. Deliverable D3.2. Initial specification and proof of concept implementation of techniques to enhance performance and resource utilization in networks. April 2015. Available online[2].
- [D42] PRISTINE Consortium. Deliverable D4.2. Initial specification and proof of concept implementation of innovative security and reliability enablers. April 2015. Available online[3].
- [D52] PRISTINE Consortium. Deliverable D5.2. Specification of common elements of the management framework. December 2014. Available online[4].
- [D53] PRISTINE Consortium. Deliverable D5.3. Proof of concept of DIF Management System. April 2015. Available online[5].
- [netem] The Linux Foundation. Linux Network Emulation, netem. Availabe online[6]
- [omnetpp-dwnld] OpenSim Ltd., OMNeT++ Releases, available online[7] .
- [tls] P.Chown, Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS). RFC 3269, June 2002, available online[8]
- [vwall] Virtual Wall description, available online[9].

[1] http://ict-pristine.eu/?page_id=37
[2] http://ict-pristine.eu/?page_id=37
[3] http://ict-pristine.eu/?page_id=37
[4] http://ict-pristine.eu/wp-content/uploads/2013/12/pristine-d52-draft.pdf
[5] http://ict-pristine.eu/?page_id=37
[6] http://www.linuxfoundation.org/collaborate/workgroups/networking/netem
[7] http://www.omnetpp.org/omnetpp/category/30-omnet-releases
[8] http://tools.ietf.org/html/rfc3268
[9] http://ilabt.iminds.be/virtualwall

A. Appendix A: RINA Stack configuration files

A.1. RINA stack general integration testing

A.1.1. Host 1

```
{
  "localConfiguration" : {
    "installationPath" : "/usr/local/irati/bin",
    "libraryPath" : "/usr/local/irati/lib",
    "logPath" : "/usr/local/irati/var/log",
    "consolePort" : 32766
  },
  "ipcProcessesToCreate" : [ {
    "type" : "shim-tcp-udp",
    "apName" : "test-tcp-udp",
    "apInstance" : "1",
    "difName" : "pInternet.DIF"
   }, {
    "type" : "normal-ipc",
    "apName" : "test3.IRATI",
    "apInstance" : "1",
    "difName" : "normal.DIF",
    "difsToRegisterAt" : ["pInternet.DIF"]
   } ],
  "difConfigurations" : [ {
    "difName" : "pInternet.DIF",
    "difType" : "shim-tcp-udp",
    "configParameters" : {
        "hostname" : "84.88.40.131",
        "dirEntry" : "1:11:test2.IRATI0:12:84.88.40.1284:2426",
        "expReg" : "1:11:test3.IRATI0:4:2426"
     }
  }, {
    "difName" : "normal.DIF",
    "difType" : "normal-ipc",
    "dataTransferConstants" : {
     "addressLength" : 2,
     "cepIdLength" : 2,
     "lengthLength" : 2,
     "portIdLength" : 2,
     "qosIdLength" : 2,
     "sequenceNumberLength" : 4,
     "maxPduSize" : 10000,
```

```
 "maxPduLifetime" : 60000
},
"qosCubes" : [ {
     "name" : "unreliablewithflowcontrol",
     "id" : 1,
     "partialDelivery" : false,
     "orderedDelivery" : true,
     "efcpPolicies" : {
          "initialATimer" : 500,
          "dtcpPresent" : true,
          "dtcpConfiguration" : {
               "rtxControl" : false,
               "flowControl" : true,
               "flowControlConfig" : {
                   "rateBased" : false,
                   "windowBased" : true,
                   "windowBasedConfig" : {
                     "maxClosedWindowQueueLength" : 50,
                     "initialCredit" : 50
                    }
               }
         }
     }
   }, {
   "name" : "reliablewithflowcontrol",
     "id" : 2,
       "partialDelivery" : false,
            "orderedDelivery" : true,
            "maxAllowableGap": 0,
            "efcpPolicies" : {
              "initialATimer" : 300,
              "dtcpPresent" : true,
              "dtcpConfiguration" : {
                "rtxControl" : true,
                "rtxControlConfig" : {
                  "dataRxmsNmax" : 5,
                  "initialRtxTime" : 1000
                },
                "flowControl" : true,
                "flowControlConfig" : {
                  "rateBased" : false,
                  "windowBased" : true,
                  "windowBasedConfig" : {
                    "maxClosedWindowQueueLength" : 50,
                    "initialCredit" : 50
           }
```

```
          }
        }
      }
  } ],
  "knownIPCProcessAddresses" : [ {
   "apName" : "test1.IRATI",
   "apInstance" : "1",
   "address" : 16
  }, {
   "apName" : "test2.IRATI",
   "apInstance" : "1",
   "address" : 17
      }, {
      "apName" : "test3.IRATI",
      "apInstance" : "1",
      "address" : 18
} ],
"addressPrefixes" : [ {
   "addressPrefix" : 0,
   "organization" : "N.Bourbaki"
  }, {
   "addressPrefix" : 16,
   "organization" : "IRATI"
   } ],
 "pdufTableGeneratorConfiguration" : {
     "pduFtGeneratorPolicy" : {
       "name" : "LinkState",
       "version" : "0"
     },
     "linkStateRoutingConfiguration" : {
       "objectMaximumAge" : 10000,
       "waitUntilReadCDAP" : 5001,
       "waitUntilError" : 5001,
       "waitUntilPDUFTComputation" : 103,
       "waitUntilFSODBPropagation" : 101,
       "waitUntilAgeIncrement" : 997,
       "routingAlgorithm" : "Dijkstra"
     }
  },
  "enrollmentTaskConfiguration" : {
     "enrollTimeoutInMs" : 10000,
     "watchdogPeriodInMs" : 30000,
     "declaredDeadIntervalInMs" : 120000,
     "neighborsEnrollerPeriodInMs" : 30000,
     "maxEnrollmentRetries" : 3
  }
```

```
  } ]
}
```

A.1.2. Router

```
{
  "localConfiguration" : {
    "installationPath" : "/usr/local/irati/bin",
    "libraryPath" : "/usr/local/irati/lib",
    "logPath" : "/usr/local/irati/var/log",
    "consolePort" : 32766
  },
  "ipcProcessesToCreate" : [ {
    "type" : "shim-eth-vlan",
    "apName" : "test-eth-vlan",
    "apInstance" : "1",
    "difName" : "110"
   }, {
    "type" : "shim-tcp-udp",
    "apName" : "test-tcp-udp",
    "apInstance" : "1",
    "difName" : "pInternet.DIF"
   }, {
    "type" : "normal-ipc",
    "apName" : "test2.IRATI",
    "apInstance" : "1",
    "difName" : "normal.DIF",
    "difsToRegisterAt" : ["110", "pInternet.DIF"]
   } ],
  "difConfigurations" : [ {
    "difName" : "110",
    "difType" : "shim-eth-vlan",
    "configParameters" : {
     "interface-name" : "eth1"
      }
  }, {
    "difName" : "pInternet.DIF",
    "difType" : "shim-tcp-udp",
    "configParameters" : {
        "hostname" : "84.88.40.131",
        "dirEntry" : "1:11:test1.IRATI0:12:84.88.40.1284:2426",
        "expReg" : "1:11:test2.IRATI0:4:2426"
     }
  }, {
    "difName" : "normal.DIF",
```

```
"difType" : "normal-ipc",
"dataTransferConstants" : {
 "addressLength" : 2,
 "cepIdLength" : 2,
 "lengthLength" : 2,
 "portIdLength" : 2,
 "qosIdLength" : 2,
 "sequenceNumberLength" : 4,
 "maxPduSize" : 10000,
 "maxPduLifetime" : 60000
},
"qosCubes" : [ {
     "name" : "unreliablewithflowcontrol",
     "id" : 1,
     "partialDelivery" : false,
     "orderedDelivery" : true,
     "efcpPolicies" : {
          "initialATimer" : 500,
          "dtcpPresent" : true,
          "dtcpConfiguration" : {
               "rtxControl" : false,
               "flowControl" : true,
               "flowControlConfig" : {
                   "rateBased" : false,
                   "windowBased" : true,
                   "windowBasedConfig" : {
                     "maxClosedWindowQueueLength" : 50,
                     "initialCredit" : 50
                    }
               }
         }
     }
   }, {
   "name" : "reliablewithflowcontrol",
     "id" : 2,
       "partialDelivery" : false,
            "orderedDelivery" : true,
            "maxAllowableGap": 0,
            "efcpPolicies" : {
              "initialATimer" : 300,
              "dtcpPresent" : true,
              "dtcpConfiguration" : {
                "rtxControl" : true,
                "rtxControlConfig" : {
                  "dataRxmsNmax" : 5,
                  "initialRtxTime" : 1000
```

```
                    },
                    "flowControl" : true,
                    "flowControlConfig" : {
                      "rateBased" : false,
                      "windowBased" : true,
                      "windowBasedConfig" : {
                        "maxClosedWindowQueueLength" : 50,
                        "initialCredit" : 50
              }
            }
          }
        }
  } ],
  "knownIPCProcessAddresses" : [ {
   "apName" : "test1.IRATI",
   "apInstance" : "1",
   "address" : 16
  }, {
   "apName" : "test2.IRATI",
   "apInstance" : "1",
   "address" : 17
      }, {
      "apName" : "test3.IRATI",
      "apInstance" : "1",
      "address" : 18
} ],
"addressPrefixes" : [ {
   "addressPrefix" : 0,
   "organization" : "N.Bourbaki"
  }, {
   "addressPrefix" : 16,
   "organization" : "IRATI"
   } ],
 "pdufTableGeneratorConfiguration" : {
     "pduFtGeneratorPolicy" : {
       "name" : "LinkState",
       "version" : "0"
     },
     "linkStateRoutingConfiguration" : {
       "objectMaximumAge" : 10000,
       "waitUntilReadCDAP" : 5001,
       "waitUntilError" : 5001,
       "waitUntilPDUFTComputation" : 103,
       "waitUntilFSODBPropagation" : 101,
       "waitUntilAgeIncrement" : 997,
       "routingAlgorithm" : "Dijkstra"
```

```
        }
      },
      "enrollmentTaskConfiguration" : {
         "enrollTimeoutInMs" : 10000,
         "watchdogPeriodInMs" : 30000,
         "declaredDeadIntervalInMs" : 120000,
         "neighborsEnrollerPeriodInMs" : 30000,
         "maxEnrollmentRetries" : 3
      }
   } ]
}
```

A.1.3. Host 2

```
{
  "localConfiguration" : {
    "installationPath" : "/usr/local/irati/bin",
    "libraryPath" : "/usr/local/irati/lib",
    "logPath" : "/usr/local/irati/var/log",
    "consolePort" : 32766
  },
  "ipcProcessesToCreate" : [ {
    "type" : "shim-eth-vlan",
    "apName" : "test-eth-vlan",
    "apInstance" : "1",
    "difName" : "110"
   }, {
    "type" : "normal-ipc",
    "apName" : "test1.IRATI",
    "apInstance" : "1",
    "difName" : "normal.DIF",
    "difsToRegisterAt" : ["110"]
   } ],
  "difConfigurations" : [ {
    "difName" : "110",
    "difType" : "shim-eth-vlan",
    "configParameters" : {
     "interface-name" : "eth1"
      }
  }, {
    "difName" : "normal.DIF",
    "difType" : "normal-ipc",
    "dataTransferConstants" : {
     "addressLength" : 2,
     "cepIdLength" : 2,
```

```
 "lengthLength" : 2,
 "portIdLength" : 2,
 "qosIdLength" : 2,
 "sequenceNumberLength" : 4,
 "maxPduSize" : 10000,
 "maxPduLifetime" : 60000
},
"qosCubes" : [ {
     "name" : "unreliablewithflowcontrol",
     "id" : 1,
     "partialDelivery" : false,
     "orderedDelivery" : true,
     "efcpPolicies" : {
          "initialATimer" : 500,
          "dtcpPresent" : true,
          "dtcpConfiguration" : {
               "rtxControl" : false,
               "flowControl" : true,
               "flowControlConfig" : {
                   "rateBased" : false,
                   "windowBased" : true,
                   "windowBasedConfig" : {
                     "maxClosedWindowQueueLength" : 50,
                     "initialCredit" : 50
                    }
               }
         }
     }
   }, {
   "name" : "reliablewithflowcontrol",
     "id" : 2,
       "partialDelivery" : false,
            "orderedDelivery" : true,
            "maxAllowableGap": 0,
            "efcpPolicies" : {
              "initialATimer" : 300,
              "dtcpPresent" : true,
              "dtcpConfiguration" : {
                "rtxControl" : true,
                "rtxControlConfig" : {
                  "dataRxmsNmax" : 5,
                  "initialRtxTime" : 1000
                },
                "flowControl" : true,
                "flowControlConfig" : {
                  "rateBased" : false,
```

```
                    "windowBased" : true,
                    "windowBasedConfig" : {
                      "maxClosedWindowQueueLength" : 50,
                      "initialCredit" : 50
            }
          }
        }
      }
  } ],
  "knownIPCProcessAddresses" : [ {
   "apName" : "test1.IRATI",
   "apInstance" : "1",
   "address" : 16
  }, {
   "apName" : "test2.IRATI",
   "apInstance" : "1",
   "address" : 17
      }, {
      "apName" : "test3.IRATI",
      "apInstance" : "1",
      "address" : 18
} ],
"addressPrefixes" : [ {
   "addressPrefix" : 0,
   "organization" : "N.Bourbaki"
  }, {
   "addressPrefix" : 16,
   "organization" : "IRATI"
   } ],
 "pdufTableGeneratorConfiguration" : {
     "pduFtGeneratorPolicy" : {
       "name" : "LinkState",
       "version" : "0"
     },
     "linkStateRoutingConfiguration" : {
       "objectMaximumAge" : 10000,
       "waitUntilReadCDAP" : 5001,
       "waitUntilError" : 5001,
       "waitUntilPDUFTComputation" : 103,
       "waitUntilFSODBPropagation" : 101,
       "waitUntilAgeIncrement" : 997,
       "routingAlgorithm" : "Dijkstra"
     }
  },
  "enrollmentTaskConfiguration" : {
     "enrollTimeoutInMs" : 10000,
```

```
        "watchdogPeriodInMs" : 30000,
        "declaredDeadIntervalInMs" : 120000,
        "neighborsEnrollerPeriodInMs" : 30000,
        "maxEnrollmentRetries" : 3
      }
   } ]
}
```