Deliverable-3.2

Initial specification and proof of concept implementation of techniques to enhance performance and resource utilization in networks

Deliverable Editor: Michael Welzl, UiO

Publication date:	30-April-2015
Deliverable Nature:	Report/Software
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:	ACC, qos-aware multi-path routing, C/U multiplexing, topological addressing, multi-layer routing
Synopsis:	This document describes the theoretical analysis, simulation and initial PoC implementation of resource-utilization and performance enhancing techniques; which address the main requirements described in D3.1.

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: Michael Welzl, UiO

i2CAT: Francisco Miguel Tarzan Lorente, Leonardo Bergesio, Eduard Grasa

IMT: Fatma Hrizi

ATOS: Miguel Angel Puente

UiO: Peyman Teymoori, Michael Welzl

CN: Roberto Riggio, Kewin Rausch

UPC: Sergio Leon Gaixas, Jordi Perello

Disclaimer

Executive Summary

In this document, "an initial specification and a proof of concept implementation" of the techniques proposed in the previous document, "draft specification", are presented. The goal is to show how the proposed techniques in the previous document can be implemented in RINA as a proof of concept, what their performance improvement is over other similar methods (if applicable), and what future directions are. The activities performed in D3.2 are centered around three main areas of i) programmable congestion control; ii) unification of connection-oriented and connectionless resource allocation in support of multiple levels of service; and iii) topological addressing to bound routing table sizes. This document also specifies the investigation results of these techniques as some initial policies in RINA.

In the first chapter, programmable congestion control, we will specify some new policies on how to implement a simple congestion control mechanism behaving like TCP. Then we will enhance it with an aggregate pushback mechanism. To compare it with other similar methods, we implemented Split-TCP, which is the most similar method in the Internet to our implementation in RINA. We show that RINA's congestion control performs similar to Split-TCP, and both outperform TCP. The second part of chapter one discusses a practical use-case of the application of RINA's programmable congestion control policies for the DC use case. Specific congestion detection and mitigation policies are proposed for the isolation of multiple tenant DIFs sharing the resources provided by the DC-Fabric DIF.

In chapter 2, two aspects of resource allocation in a DIF are explored: multiplexing and exploiting multiple equal-cost paths. The first part of chapter 2 discusses the Cherish/Urgency multiplexing scheme, by which it is possible to provide strict differential delay and loss guarantees to different flows provided by a DIF. We first present this multiplexing approach in the context of the more general ΔQ framework. Next we analyze how this multiplexing approach can be decomposed into the different RMT policies, propose a specification for those policies and perform an initial performance evaluation using RINAsim. The second part of the chapter is dedicated to exploring a QoS-aware multi-path routing approach in the context of the datacentre networking use case. The

fact that a DIF is aware of the QoS requirements of the flows provided to the applications using the DIF opens the door to more effective and dynamic multi-path strategies than the traditional Equal Cost Multi Path (ECMP) approach.

In chapter 3, Topological Addressing to Bound Routing Table Sizes, we present new generic architectures for routing and addressing tailored to cope with the different PRISTINE use cases requirements in terms of scalability, reliability and efficiency. Moreover, these architectures are designed to benefit from RINA’s recursive architecture. A couple of routing policies have been implemented and tested along with the proposed architectures in order to examine the performance of applying RINA to the use cases, specifically the distributed clouds use case. Simulation results show that a significant improvement of the routing table size is achieved.

Table of Contents

List of Figures

List of acronyms

ACC	Aggregate Congestion Control
AE	Application Entity
AI	Application Instance
AP	Application Process
CACEP	Common Application Connection Establishment Protocol
CCP	Continuity Check Protocol
CDAP	Common Distributed Application Protocol
DA	Distributed Application
DAF	Distributed Application Facility
DC	Data Centre
DIF	Distributed IPC Facility
DTCP	Data Transfer Control Protocol
DTP	Data Transfer Protocol
E2E	End to End
ECN	Explicit Congestion Notification
EFCP	Error Flow Control Protocol
FA	Flow Allocator
FAI	Flow Allocator Instance
FIFO	First In, First Out
FQ	Fair Queuing
IANA	Internet Assigned Numbers Authority
IPC	Inter Process Communication
IRM	IPC Resource Manager
ISP	Internet Service Provider
LAN	Local Area Network
LIFO	Last In, First Out
MAC	Medium Access Control
NM-DMS	Network Management Distributed Management System
MPLS	Multi-Protocol Label Switching
MPLS-TE	MPLS with Traffic Engineering extensions
NSM	Name-Space Manager
OS	Operating System
OSPF	Open Shortest Path First
PCI	Protocol-Control-Information
PDU	Protocol Data Unit

PDUFG	PDU Forwarding Generator Policies
PFT	Protocol Data Unit Forwarding Table
PFTG	PDU Forwarding Table Generator
PoA	Point of Attachment
QoS	Quality of Service
RA	Resource Allocator
RIB	Resource Information Base
RINA	Recursive InterNetwork Architecture
RIR	Regional Internet Registry
RMT	Relaying and Multiplexing Task
RR	Round Robin
RSVP-TE	ReSerVation Protocol with Traffic Engineering extensions
SDU	Service Data Unit
SFR	Scalable Forwarding in RINA
TCP	Transmission Control Protocol
WLAN	Wireless LAN

1. Congestion control

1.1. Programmable Congestion Control

1.1.1. Introduction

Congestion control is done by the TCP protocol in the Internet in an "end-to-end" fashion. The sender is responsible of recovering lost packets, guessing the right sending rate, and avoiding from congesting the network. This ends up in problems such as ignorance of the underlying technology and lengthening the time-to-notify of the sender, which result in losing efficiency. RINA, with its strong layering, allows to attain efficiency in a divide-and-conquer manner, which enables usage of more elaborate mechanisms inside the network wherever they do fit.

We specify how congestion control has been implemented in RINA. By the recursive nature of RINA, its congestion control is bound to one DIF, making a series of DIFs operate in a fashion that is similar to split-TCP (in case the per-DIF congestion control policy is TCP-like). First we introduce RINA congestion control policies, and then, we present simulation results compared with TCP and Split-TCP [SplitTCP].

1.1.2. RINA ACC Policies

Here, we specify the policies implemented/used for ACC in RINA. A simple DIF in shown in Figure 1 in which there are a sender, a relay, and a receiver IPC processes.

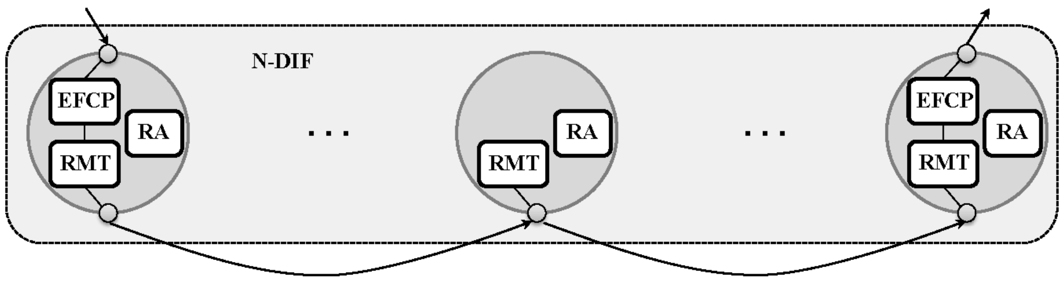


Figure 1. A simple DIF

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/or RMTMaxQPolicy. Policy implementations:

  - *REDMonitor*: Using this implementation, the average queue length is calculated and passed to REDDropper.

- **RMTMAXQPolicy**: This policy is invoked when a queue reaches its threshold or exceeds its maximum queue length 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*: Using this implementation, if the output queue size of RMT exceeds a maximum threshold when inserting a new PDU to an output queue, the RMT requests the Resource Allocator (RA) to send 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 the initial threshold when inserting a new PDU to the queue but it is shorter than the max threshold, RMT requests the Resource Allocator (RA) to send a direct congestion notification to the sender of that PDU using a CDAP message with a probability proportional to the amount exceeding the initial threshold.

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*: This is a TCP-like congestion control implementation in RINA. 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 (via the Resource Allocator), 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.

  - *DTCPTxControlPolicyTCPWireless*: This implementation is used for the DIFs functioning over a wireless link where a packet loss does not always imply congestion.

- **DTPRTTEstimatorPolicy**: This policy is used to calculate Round-Trip-Time (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 RINA upon receiving ACKs, and further it 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 sent by the Resource Allocator of a congested relay node to the IPC Process of the sending EFCP of a PDU. Clearly, both IPC Processes are in the same DIF. The notification is sent as a CDAP message. In other words, this is the corresponding policy triggered by UpstreamNotifier in the sending EFCP. Policy implementations:
  - *TCPECNSlowDownPolicy*: Referring to Algorithm 1, this policy only calls the OnRcvSlowDown procedure of DTCPTxControlPolicyTCPTahoe in case of using DTCPTxControlPolicyTCPTahoe as DTCPTxControlPolicy.

Algorithm 1 Pseudo code of DTCPTxControlPolicyTCPTahoe

```
procedure INITIALIZE
    cwnd $\leftarrow$ RST_WND
    ssthresh $\leftarrow$ MAX_SSTHRESH
    slowedDown $\leftarrow$ false
    state $\leftarrow$ STATE_SLOW_START
    sendCredit $\leftarrow$ cwnd
    flightSize $\leftarrow 0$
end procedure
procedure SEND
    if sendCredit $> 0$ then
        FCCredit $\leftarrow$ FlowControl.getCredit()
        dtcpState.pushToPostablePDUQ(min(sendCredit, FCCredit))
        numOfSent $\leftarrow$ dtcpState.getNumOfSentPDUs()
        flightSize $\leftarrow$ flightSize + numOfSent
        sendCredit $\leftarrow$ sendCredit - numOfSent
        slowedDown $\leftarrow$ false
        if sendCredit $= 0$ then
            dtcpState.setClosedWindow(true)
        end if
    end if
end procedure
procedure ONRCVACKS(int ackNum)
    flightSize $\leftarrow$ flightSize - ackNum
    inc $\leftarrow$ ackNum
    if inc $> 3$ then
        inc $\leftarrow 3$
    end if
    if state = STATE_SLOW_START then
        cwnd $\leftarrow$ cwnd + inc
    end if
    if state = STATE_CNG_AVOID then
        cwnd $\leftarrow$ cwnd + inc / cwnd
    end if
    if (cwnd $\geq$ adv_wnd) or (cwnd $\geq$ ssthresh) then
        state $\leftarrow$ STATE_CNG_AVOID
    end if
    sendCredit $\leftarrow$ round(cwnd) - flightSize
    if sendCredit $> 0$ then
        dtcpState.setClosedWindow(false)
    end if
end procedure
procedure ONRCVSLOWDOWN
    if not slowedDown then
        state $\leftarrow$ STATE_CNG_AVOID
        cwnd $\leftarrow$ min((cwnd / 2), 2)
        ssthresh $\leftarrow$ cwnd
        slowedDown $\leftarrow$ true
    end if
end procedure
procedure ONRCVDUPACK
    OnRcvSlowDown()
end procedure
procedure ONTIMEOUT
    state $\leftarrow$ STATE_SLOW_START
    ssthresh $\leftarrow$ min((cwnd / 2), 2)
    cwnd $\leftarrow$ RST_WND
end procedure
```

Figure 2. Algorithm 1

Algorithm 2 Pseudo code of DTPRTTEstimatorPolicyTCP

```
 1: procedure INITIALIZE
 2:     state ← STATE_FIRST
 3:     k ← 4
 4:     G ← 0.1
 5:     $\alpha$ ← 0.125
 6:     $\beta$ ← 0.25
 7: end procedure
 8: procedure RUN(int seqNum)
 9:     newRTT ← now() - getSentTime(seqNum)
10:     if state = STATE_FIRST then
11:         RTTVar ← newRTT / 2
12:         SRTT ← newRTT
13:         state ← STATE_NEXT
14:     else if state = STATE_NEXT then
15:         RTTVar ← $(1-\beta)*$ RTTVar $+\beta*$ABS(SRTT − newRTT)
16:         SRTT ← $(1-\alpha)*$ SRTT $+\alpha*$newRTT
17:     end if
18:     RTO ← RTT + max(G, k * RTTVar)
19:     if RTO < 1 then
20:         RTO ← 1
21:     end if
22:     dctpState.setRTT(newRTT)
23:     dctpState.setRTO(RTO)
24: end procedure
```

Figure 3. Algorithm 2

1.1.3. Discussion on the Use of ACC Policies

It should be noted that different combinations of the above policies can be used in a network. Furthermore, different layers might use different kinds of policies, or the polices are chained among layers to create a chained pushback up to the root cause of congestion. A sample general architecture of N layers is illustrated in Figure 4. The dashed, curved arrows in the figure represent notification transmission between modules. Assume that we only use UpstreamNotifier in the RMTs in the network. In case of congestion in 1-DIF, the Resource Allocator (RA) of the transmitting IPC Process sends a notification to the IPC Process of the EFCP instance sending the PDU, the EFCP instance is located in the same IPC process. The congestion might end up in the closed window queue length growth in the EFCP instance. If the closed window queue length reaches its maximum size limit, the EFCP shuts down it incoming port, and PDUs are backlogged in the upper RMT output queue. The port is unblocked if the queue length decreases. The output queue of the RMT in 2-DIF might reach its maximum threshold. Since this RMT also uses UpstreamNotifier, this process continues; it sends a slow down signal to the RA and the RA sends a notification to the sending EFCP instance. This pushback notification continues until it reaches the source of congestion. It is worth noting that the EFCP instance transmitting the traffic at each layer might be carrying the traffic of an aggregate of upper-layer flows which have been mapped to one lower-layer flow. This is why we call it Aggregate Congestion Control (ACC).

The combination of the above policies can facilitate handling of some complicated situations. For example, in a wireless link, a packet loss may not be a sign of congestion, but a backlogged output queue over time is. By considering this fact, a DTCPTxControlPolicy policy such as DTCPTxControlPolicyTCPWireless described above can be developed that halves its cwnd only when it receives a notification from an UpstreamNotifier policy; if it receives, for example, triple duplicate acknowledgments, it does not halves its cwnd as the normal TCP does.

Since sending a direct notification to the sending EFCP instance might affect scalability, especially when the upstream path towards the sender/s is loaded. This depends on the network topology and the usage scenario. In this case, other policies such as REDUpstreamNotifier, ECNMarker, or REDDropper should be used.

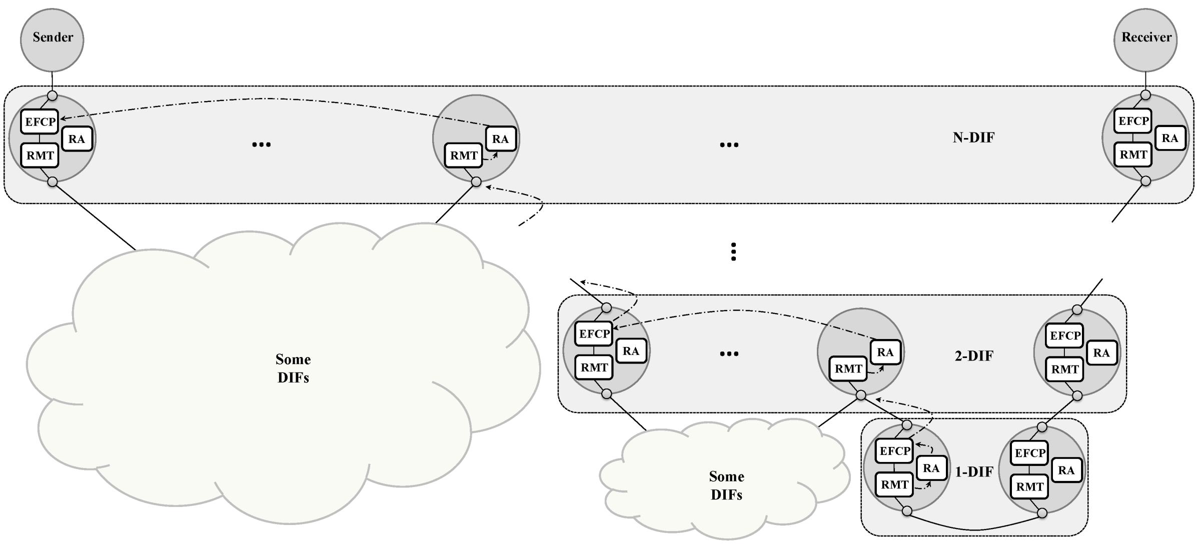


Figure 4. Layered upstream notification in RINA

1.1.4. Evaluation: Simple ACC

RINA has been implemented in the state-of-the-art discrete event simulation tool OMNet. This implementation provide a varieties of policies with default behavior which can be customized or extended for developing new protocols and applications. In this section, we present simulation results of ACC in RINA compared with TCP and Split-TCP. We implemented Split-TCP in the INET framework of OMNet.

Comparison of RINA-ACC with TCP and Split-TCP

The simplest topology for using Split-TCP is shown in Figure 5; there is a router in the path from sender S_1 to receiver R_1. This router acts as a splitter meaning that upon the receipt of a TCP data segment, it sends an acknowledgment to the sender, and then using another TCP connection which might use a different TCP flavor, e.g. TCP variations customized for wireless links, it forwards the segment to the destination. In case of any packet loss in the second transmission, the router retransmits the lost packet without involving the sender. Moreover, if the second link between the router and the receiver is congested temporarily, the splitter can reduce its sending rate faster.

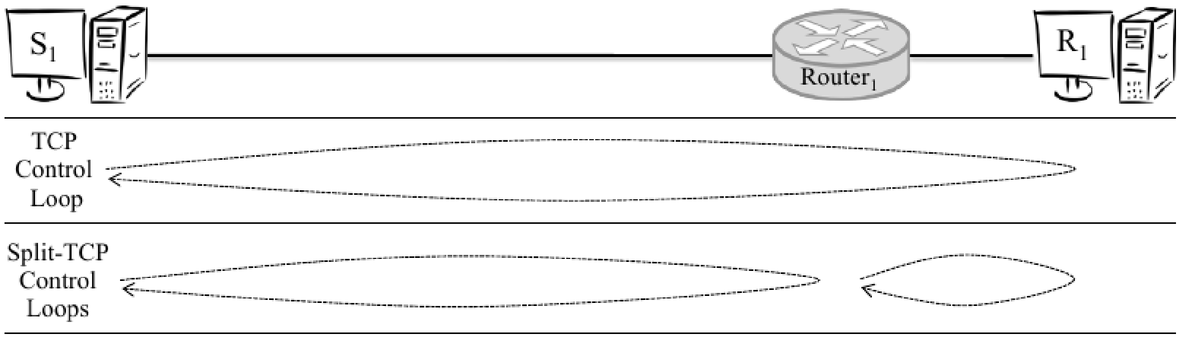


Figure 5. The network topology used for comparison

EFCP instances in DIFs act similarly; they locally handle situations such as packet loss or sending rate adjustment inside one DIF. In order to compare RINA-ACC with Split-TCP and the performance gain over TCP, we used the network topology illustrated in Figure 5. In the first simulation scenario, only S_1 sent data to R_1. The traffic sent by S_1 was a large file so there is only one flow from S_1 to R_1 in the network. The simulation scenario was run for a specific duration, 1 minute, and after that we collected measures such as the total volume of data transmitted. The bandwidth of the link between the sender and the router is 10Mbps, and the link between Router$_1$ and R_1 is 2Mbps. The output queue size of the network interfaces of all nodes is limited. In case of using splitter in Router$_1$, an internal, infinite-size buffer was assumed for the splitter.

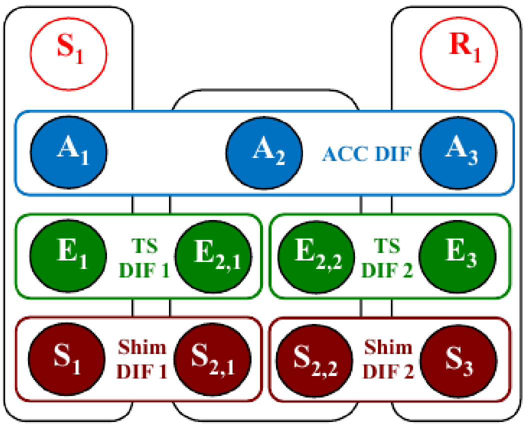


Figure 6. ACC in RINA

In case of using RINA-ACC whose architectural view is also shown in Figure 6, DTCPTxControlPolicyTCPTahoe was activated for both EFCPs in IPC processes $E_{2,2}$ and A_1. In addition, the UpstreamNotifier policy was set for the RMTs in IPC processes $E_{2,2}$ and A_2. This configuration of pushback means that if congestion happens between the router and the receiver, first the RMT in $E_{2,2}$ detects it through observing its output queue length by UpstreamNotifier; if the queue length exceeds its initial threshold, the RMT sends a pushback notification to the sender of the last PDU pushed into that queue. Since $E_{2,2}$ is in the TS DIF 2, the sender is the EFCP inside $E_{2,2}$ which receives the pushback notification and slows down. If there are other SDUs waiting for this EFCP to be transmitted and the congestion window does not allow it, they are backlogged in the closed window queue of the EFCP. If the relayed PDUs in A_2 have a higher rate, then the closed window queue length of the EFCP in $E_{2,2}$ is built up; upon reaching its maximum limit, the EFCP stops receiving more SDUs from the upper RMT in A_2. In particular, EFCP shuts down its incoming port until the closed window queue has empty space which as a result, unblocks the port. If the relay rate of A_2 is still high, the length of its output queue towards $E_{2,2}$ grows until it reaches its maximum threshold. Since this RMT uses UpstreamNotifier too, in this situation, it sends a direct CDAP notification to the EFCP instance of the sender of the last PDU pushed into the queue. At this layer, the notification receiver is the EFCP instance of A_1 which is located in the sender node.

In case of using TCP, since the second link is slower and it is going to be saturated soon which results in packet drop in the output link of the router towards the receiver, the sender will be notified implicitly by receiving duplicate acknowledgments. The TCP flavor used in the TCP and Split-TCP cases was TCPReno with Selective Acknowledgment (SACK) enabled.

Figure 7 shows the congestion window size of RINA-ACC, Split-TCP, and TCP for the above scenario where the link delay between the sender and the router is 75ms, and 25ms between the router and the receiver which result in a total Round-Trip-Time (RTT) of 200ms without any queuing delay. We see that cwnd of TCP increases until it receives a decrement notification. Due to the longer time-to-notify duration, it increases to much higher or lower values of the optimum rate at which it should be sending. Instead, RINA-ACC and Split-TCP, RINA-ACC:Relay and Split-TCP:Splitter in the figure, react faster to such notifications. Since the link was slow during the whole simulation time, the sender had to reduce its rate. We can see that cwnd of RINA-ACC:Sender decreases as well at two time instants.

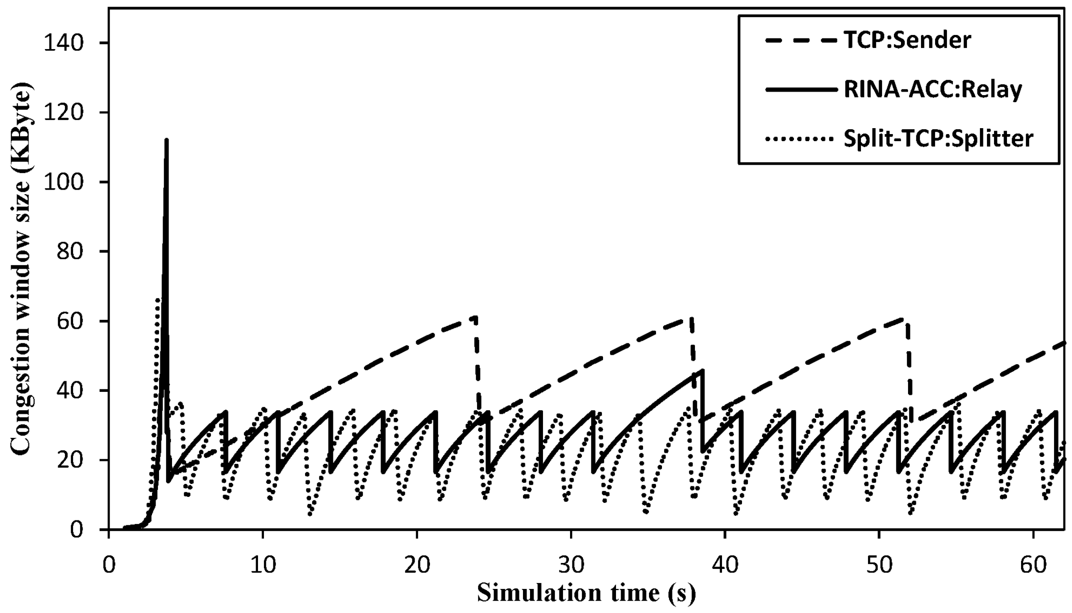


Figure 7. Congestion window size

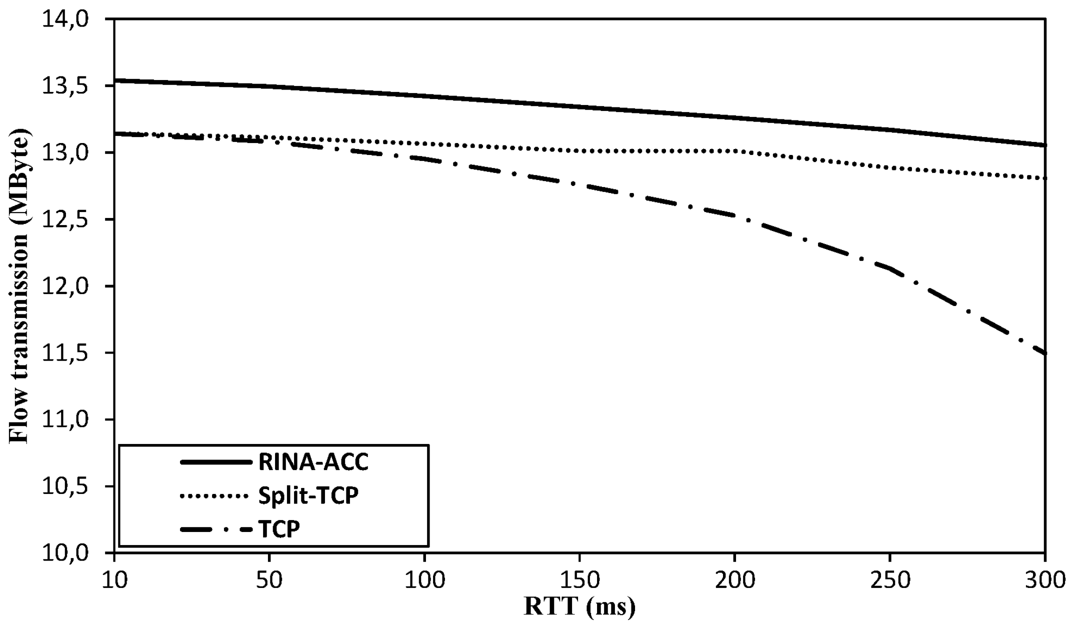


Figure 8. Transmitted volume of data

The performance of these three methods is illustrated in Figure 8. The vertical axis shows the total volume of data transmitted in 1 minute over different RTT values. Three forth of each RTT value is spent in the link between the sender and the router. Shorter RTTs, as shown in the figure, do not affect the performance of the three methods. However, longer RTTs increase time-to-notify especially in TCP which cause performance degradation. Split-TCP and RINA-ACC perform almost the same for longer RTTs, but there is a little performance improvement in RINA-ACC because instead of letting the packets be dropped which incurs retransmission and wasting the bandwidth, it sends earlier explicit congestion notifications using pushback to prevent this.

Multiple Flows

We simulated the situation in which there were multiple flows in the network. In the first scenario, two flows shared the same bottleneck link in RINA. The network connectivity graph is shown in Figure 9; senders S_1 and S_2 send a large file to receivers R_1 and R_2, respectively. All the links had the same bandwidth. This means that the link between $Router_1$ and $Router_2$ was a bottleneck link. There was one lower-layer DIF per each link, and one upper-layer DIF on top of them. The RMTs used UpstreamNotifier, and DTCPTxControlPolicyTCPTahoe was used as the congestion control policy. After running the simulation for 1 minute, we observed that both receivers had received almost the same volume of data meaning that the bottleneck link had been shared equally between the two senders, and the

UpstreamNotifier policy in the upper-layer DIF of $Router_1$ had sent almost the same amount of pushback notifications to the senders.

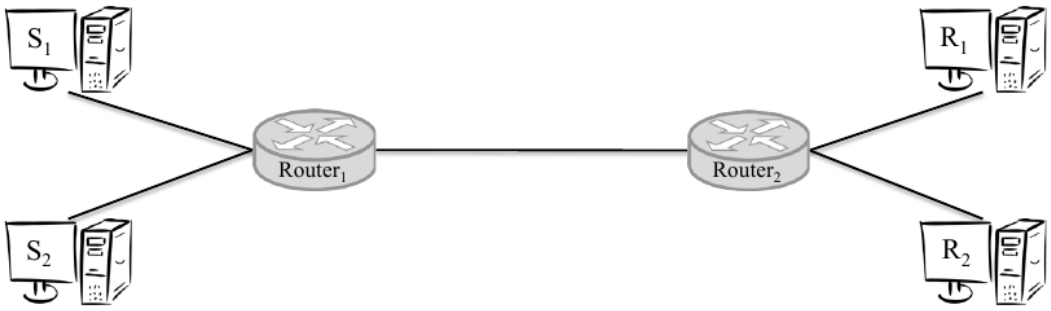


Figure 9. Network topology for two flows

In the next simulation scenario on this network topology, the link between $Router_2$ and R_2 had a lower bandwidth than the others, and the bandwidth of the link between $Router_1$ and $Router_2$ was twice of that of the others. In this case, the UpstreamNotifier policy in the upper-layer DIF of $Router_2$ had sent notifications only to S_2, and S_1 had been limited by its bandwidth limit of its link towards $Router_1$. Since in this scenario, the network throughput results for different values of RTT behave the same as those depicted in Figure 8, we ignored drawing a new diagram for it. However, these results also confirm the success of RINA-ACC in improving performance.

1.2. Policies for performance isolation in multi-tenant data centres

1.2.1. Introduction

In this section we describe the design of a set of policies addressing the performance isolation problem in multi-tenants data-centres. In this context, tenants purchase computing, storage, and networking resources from a cloud operator. As such, a tenant would expect a virtual network to provide the same usage experience as one would expect when operating the same resources on a dedicated physical network deployed at the customer's premises. With respect to the network fabric this corresponds to ensure the isolation of the network slices assigned to each tenant.

The design of the policies presented in this section is inspired by [EyeQ], [Riggio] however as opposed to the original works, where a considerable number of hacks and ad hoc solutions had to be devised, we shall see how the recursive and modular nature of RINA will allow to achieve similar goals by using a small number of re—usable and general purpose policies. In order to guarantee performance isolation in multi-tenants networks our solution leverages on two technical features: (i) bandwidth requests made by the tenants are enforced at the edges, and (ii) a feedback about the actual network utilization must be provided to source of each flow.

The mechanism at the base of the notification mechanism is the Explicit Congestion Notification flag, i.e. the PDUs of the congested flows are marked in a way that allows the receiver instance to detect that a congestion is actually taking place. Once the receiver knows about the congestion, it can apply various strategies in order to mitigate it. The reaction policy presented in this section will tune the flow rate to reduce the transmission rate and adapt it to the optimal rate necessary to maintain the link at its maximum usage rate without incurring in congestion.

1.2.2. Data-centre organization assumptions

Figure 10 illustrates the organization of the different DIFs in the DC use case, as described in [D21]. The *DC fabric DIF* unifies all the DC resources in a single resource pool, supporting multiple *tenant DIFs* that get a fraction of the DC resources allocated for their use. In this discussion we assume that the tenant DIFs are self-contained in a single DC.

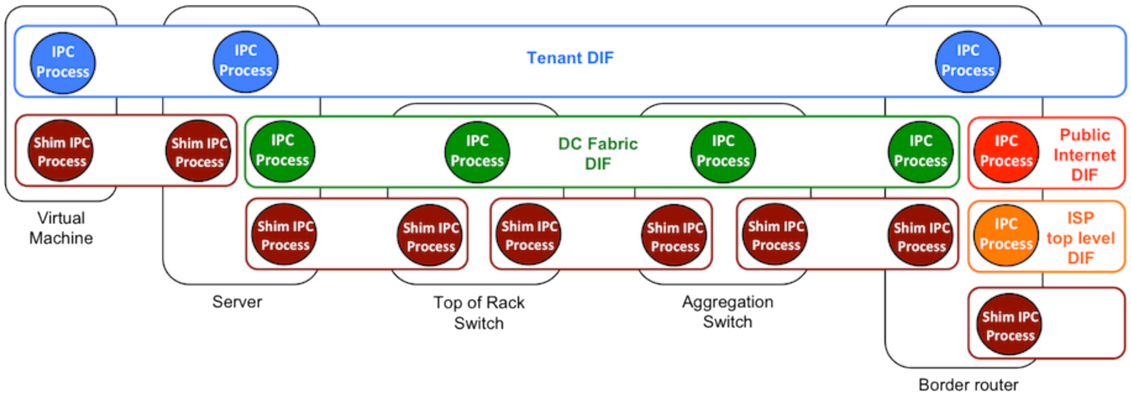


Figure 10. DIF architecture for Datacenter Networking

Full-bisection bandwidth

In a network that is fully under the control of the cloud operator, solutions capable of exploiting full bisection bandwidth topologies available in modern data-centres [Niranjan], [Guo], [Greenberg] do exist. Examples include the Equal–Cost Multi–Path protocol [Hopps] and Hedera [AlFares].

The development of a routing policy capable of exploiting the availability of a full-bisection bandwidth topology such as ECMP is discussed in section 2.2.

Minimum granted bandwidth

The **guaranteed bandwidth** is the minimum guaranteed speed at which each distinct pair of IPC Processes in the Tenant DIF can communicate. If, for example, two different IPC processes (using of the same DIF) have a common destination IPC process, and both sources communicate at the given minimum granted bandwidth (MGB), then the destination IPC process will have an incoming total traffic of 2 x MGB. If such an amount exceed the physical link capacity, then the source IPC have to reduce their transmission rate. Figure 11 illustrates this situation with an example: if the link maximum capacity is 'n', then the DIF can access the full capacity, and all the available resources can be used. In this example the Blue DIF, having a MGB of 7 is accessing the full capacity of 10.

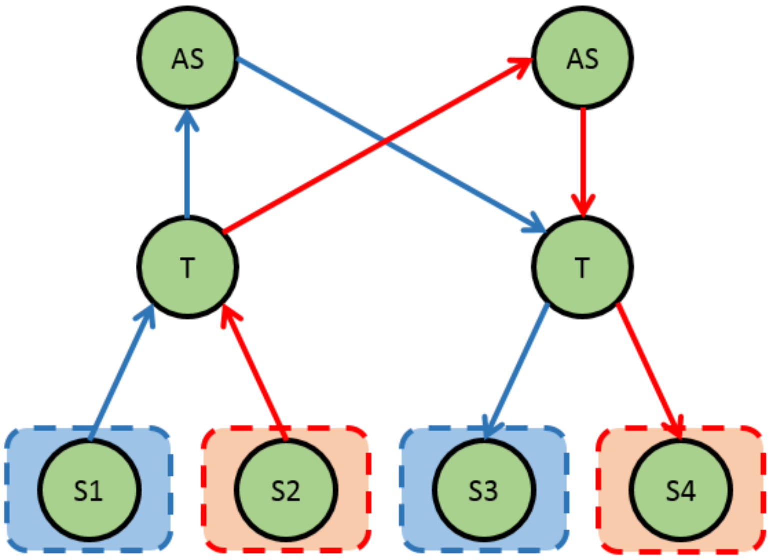


Figure 11. DC Fabric DIF providing two flows to tenant DIFs (red, blue)

No bandwidth oversubscription

Another assumption is that, during the Tenant DIF creation stage, the Network Manager will make sure that the aggregate minimum bandwidth allocated to the tenants using a certain server does not exceed the nominal link capacity, i.e. no link oversubscription.

If, for example, the link connecting a node to its Top of Rack switch is a 1 Gb/s link, then the sum of the Minimum Granted Bandwidth of the Virtual Machine present on such node must be equal to or smaller than 1 Gb/s. As shown in Figure 12, if the link capacity of the POD is limited to 10, then the sum of the minimum granted bandwidth of the DIFs using a node must not exceed it. In this case the sum of the MGB of the 3 DIFs present in S1 is 12, and so the last DIF (the yellow one) will be rejected (at the moment of the creation request) or allocated in a different node (Figure 13).

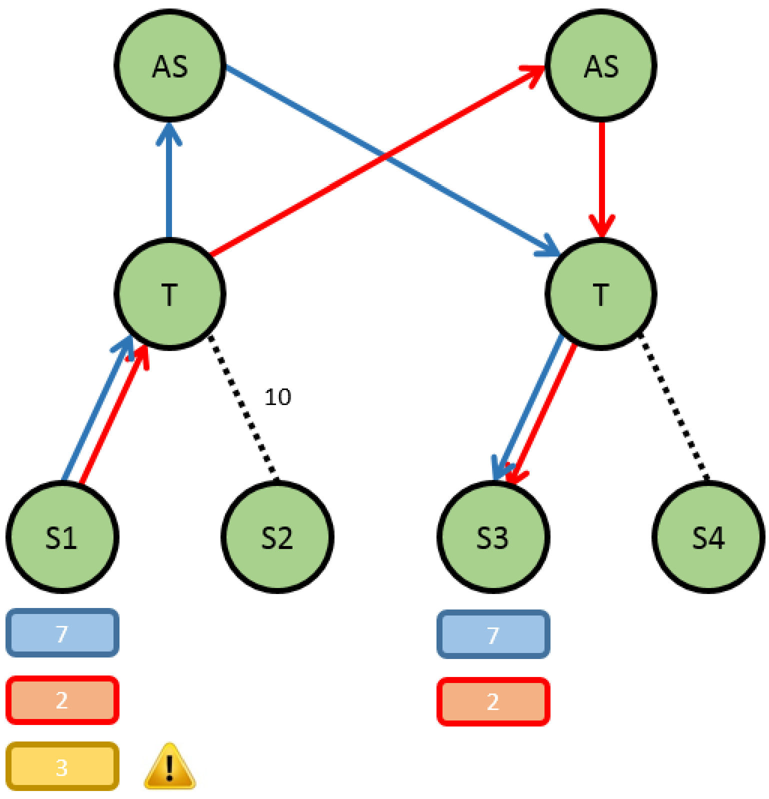


Figure 12. Over subscription of the VMs minimum granted bandwidth

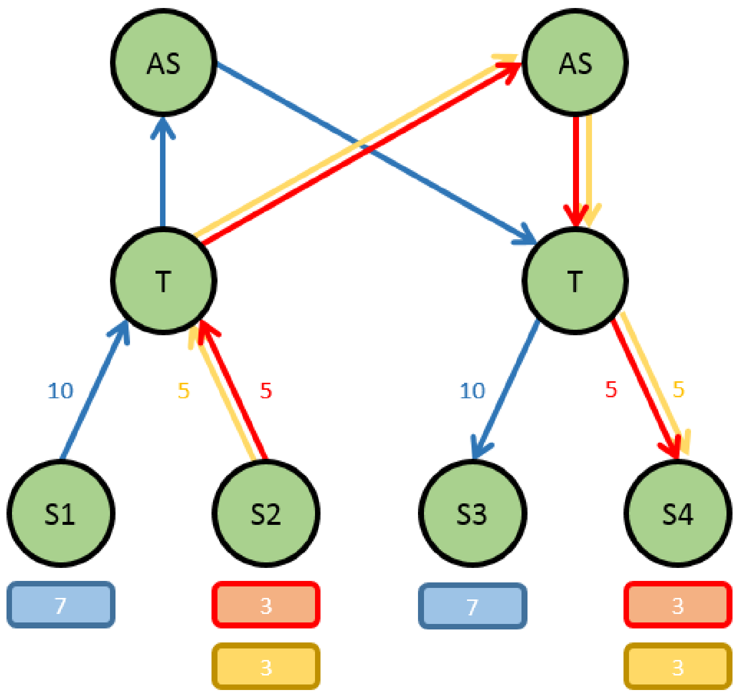


Figure 13. Reallocation of VM to non-oversubscribed node

Use all the available bandwidth

Flows created using these policies will begin their communication trying to use the full available bandwidth. If the VM is the only one present on the node, it will possible for it to exploit the full line rate of the link. If other VMs are present in the Server node, they will be sharing the bandwidth according to their bandwidth requests.

Congestion caused by this behavior will be handled by the RDSR policies which, depending on the strategy selected, can decide to reduce the bandwidth respecting the minimum granted bandwidth constrain.

1.2.3. Policies for Congestion Control

In order to perform Congestion Control, we will need a mechanism which detects when a congestion is happening, and a mechanism which reacts to the congestion in order to solve it.

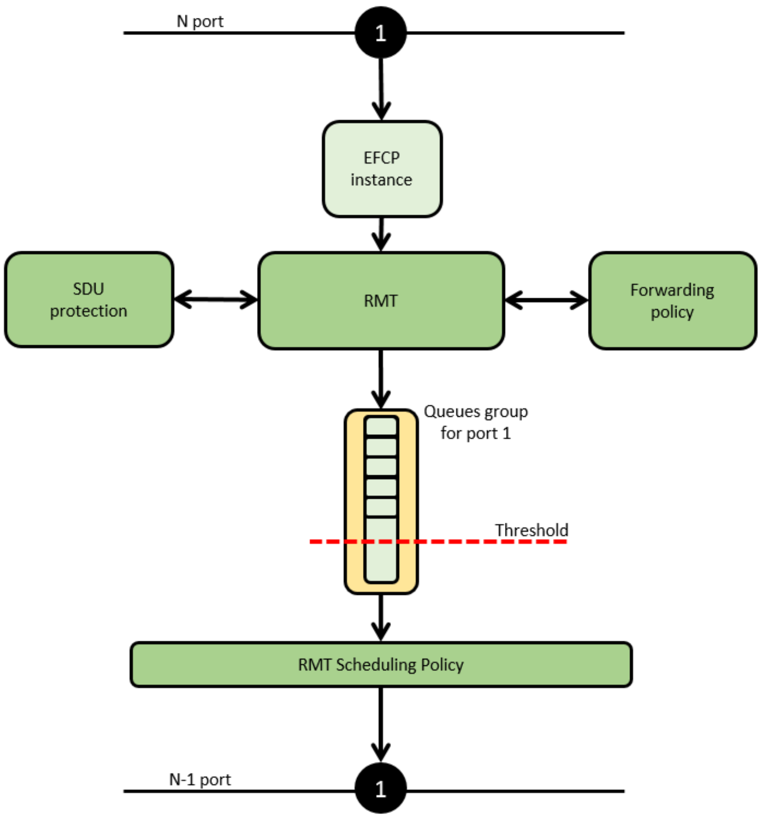


Figure 14. RMT policy for CC

The **Congestion Detection** mechanism will be fulfilled by a RMT policy: this policy will monitor incoming and outgoing packets, evaluating the state of its queues. If the load of the queues becomes higher than an assigned threshold, then the port enters in a congested state. When in this state, the outgoing PDUs scheduled on that port will be marked as congested, and will carry such information to the destination EFCP instance. Conversely, when the status of the port return to an acceptable level, i.e. the load of the queue is smaller than the assigned threshold, the congested state is revoked and the outgoing PDUs are not marked anymore with the congestion flag. This scheme is depicted in Figure 14.

The **Congestion Reaction** mechanism will be fulfilled by an EFCP policy: this policy will control incoming PDUs looking for the congestion flag. If such flag is detected, this means that a congestion is taking place somewhere in the network, and this flow could partly cause of such congestion. The EFCP instance will then use the rate control mechanisms offered by the SDK; this will cause the flow to limit it’s transmission rate to a pre-computed rate, as shown in Figure 15. The EFCP policy detects the congestion at point 1, and marks passing PDUs with the congestion flag. When such packets reach the EFCP (point 2) then the rate reduction mechanism is invoked in order to reduce the flow transmission rate (point 3). This in time will trigger the IPCP in the upper DIF to apply their own congestion policy in order to resolve it (point 4). This process continues then recursively (or at least unless the congestion policy say so) on the upper layers

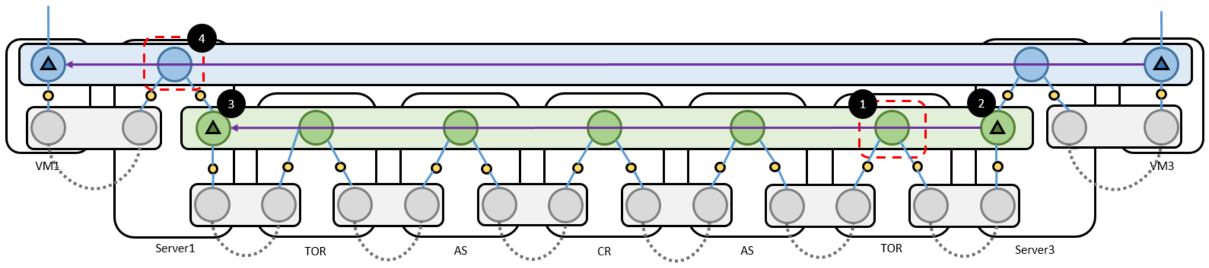


Figure 15. Decreasing the sender rate using flow control

In order to better tune such feedback message for the EFCP source instance, more knowledge about the EFCP instances is needed. This is due to the fact that, simply reducing the transmission rate to a given pre-computed value (the minimum guaranteed bandwidth), could lead to an under utilization of the network resources. This additional knowledge will be provided by a kernel implementation of the Resource Allocator. Such functionalities will allows the EFCP instances to exchange information about their current transmission rates.

1.2.4. Implementation steps

The two Congestion Control policies will be developed independently one from another. The roles of the policies are well defined and do not overlap between themselves: the congestion detection will reside in the RMT and will mark the PDUs using the Explicit Congestion Notification while the EFCP policy will react to ECN marked packets (which can be marked also by other policies, if someone need to develop a different congestion notification mechanism) and apply a rate control over the flow affected by congestion.

Notice how the RMT policy can be used in all scenarios where it is necessary to react to potential congestion events. Similarly, the EFCP congestion mitigation policy devised in this section can be used separately from the RMT policy in order to enforce a certain transmission rate for a flow. The side-effect of this design choice is that the two policies can be effectively reused by project partners or third parties, maximizing code reuse.

We envision the following roadmap for the implementation phase:

The first step will be to implement the RMT policy which has to detect congestion occurring in the network. Such policy will organize the queues per port, and will monitor their states in order to avoid the queue to exceed a threshold (which will be defined statically or be computed dynamically based on the maximum queue size). Once the threshold for a queue is exceeded, then the port enters in a congested state: traffic outgoing from this port will be marked as congested using the ECN flag in the PDU's header. If the queue returns back to acceptable levels (under the defined threshold) then the port exits from the congested state, and PDUs are no longer marked as congested.

This policy will be initially tested on a very simply topology, composed by 3 nodes, as shown in Figure 16. The first will be connected to the second, and the second to the third by a single N-1 flow. An application from the first node will communicate with an application present on the third node. The second node (in the middle) will have a strict threshold so the communication between first and third will immediately generate a congestion. The communication on the third node will be then analyzed in order to detect the ECN flag turned on when the second is manually pushed in a congested state.

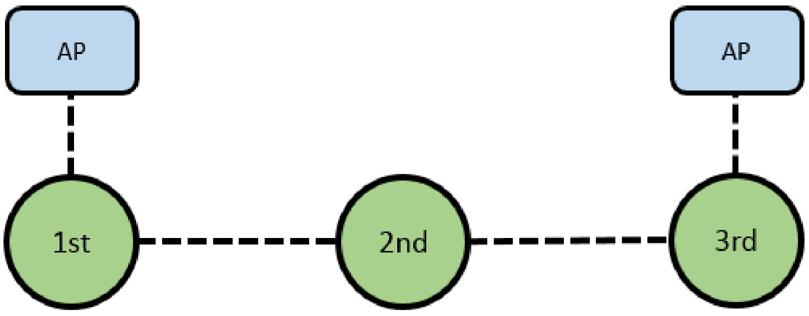


Figure 16. Simple test for CC policies

The second step will be introducing an EFCP policy which reacts to ECN marked incoming PDUs. When a packets with such characteristic is detected, then the EFCP knows there's a congestion in the network. Such policy will impose a different transmission rate to the flow in order to lower the network loads.

This policy will also be initially tested on the same simple topology as seen before. Once the third node receives the ECN-marked PDU then it will use the flow control mechanism offered by the stack to reduce the flow transmission rate. The first node will be analyzed in order to detect the rate reduction command and effects.

The last step will take place when a base testing on the policies ensure that they behave as described by the design requirements. The DC experiment will be swapped-in in the VWall testbed as described in [D61], and the Congestion Control policies will be assigned at the DC-DIF layer. A series of experiments will be then run on the top of Tenant DIF spawned over DC DIF in order to cause congestion and measure the reaction of the IPCPs in the DIF to it.

2. Resource Allocation

2.1. Traffic differentiation via delay-loss multiplexing policies

This section analyzes two distinct approaches for scheduling and forwarding different types of traffic within a DIF, primarily focusing on the delay and loss requirements of the flows, while drafting policies for the relevant RINA components to this task.

2.1.1. Flow cherish and urgency

In scheduling, assigning a higher priority to a flow implies serving its packets before than others', thus keeping its delay lower than that of flows with lower priority. In a similar way, having a higher threshold on compound queue occupation (number of packets on all the queues that goes to the same port) ensures, in a congested environment, that a flow will start to have losses due to full buffers later than flows with lower thresholds.

While these priorities and thresholds do not provide a fair distribution of resources between the distinct flows, they provide an easy and fast way to perform flow differentiation given their urgency (requirement of low delay) and cherish (requirement of having low losses). If we consider these two parameters separately and not strictly related as in traditional scheduling approaches (like in Weighted Fair Queuing, WFQ), finer distinction of QoS classes can be achieved. For example, we could effectively support QoS classes desiring low delays but allowing more losses, others targeting lower losses but allowing some additional delay, etc. Such a distinction between delay and loss offers an additional degree of freedom to the scheduling discipline, and thus providing better discrimination between different flow requirements in terms of QoS.

Working toward an efficient scheduling discipline to be incorporated in RINA IPC processes that effectively accounts for both delay and losses, in this deliverable we firstly analyze the functionalities that would have to be incorporated to implement the ΔQ approach [Davies], as well as the specification of the IPCP policies that would have to be programmed. As we will highlight, the complete ΔQ approach entails significant complexity. Hence, we leave its full implementation for the second iteration of the PRISTINE Project, while in the first one we contemplate simplified Delay/

Loss and Enhanced Delay/Loss scheduling schemes. As we show later on by simulation results, such scheduling schemes effectively provide delay and loss differentiation among the supported QoS classes, and thus will be prototyped as an initial QoS differentiation solution for RINA DIFs using the PRISTINE SDK. Then, in the second iteration of the project, these solutions will be subsequently empowered, so as to finally behave as defined in the complete the ΔQ approach.

2.1.2. The ΔQ approach to QoS

ΔQ is a scheduling model proposed by the company Predictable Network Solutions, based on the idea that a triad of parameters (bandwidth usage, delay and losses) determine the behavior of flows and, by fixing one of these parameters (bandwidth typically), it is possible to provide efficient treatment and predictable QoS experience to the others. The ΔQ scheduling module consist in a set of sub-modules (see the Figure below for a simplified view):

- **Queue Manager**: It is the queue management module that manages the usage of available buffers, maintains the distinct queues (using pointers to those buffers) and ensures that data on buffers is maintained until sent to all required ports (sending packets on multicast only requires one buffer per packet) or dropped.
- **Policer/Shaper (P/S)**: This module has two different functionalities. It ensures that flows do not exceed the resource utilization contracted in their service level agreement and also shapes them in order to smooth transient data bursts. Specifically, on the policer side, it controls the maximum bandwidth usage for each flow and selects the cherish/urgency for each packet, in order to ensure the requirements of the flow. On the shaper side, a small/random delay is added between the packets of a flow in order to smooth burst of data while avoiding the starvation of low urgency flows.
- **Flow Selection**: The flow selection module is responsible for forwarding packets to the distinct P/S depending on their headers. At this point, packets can be dropped (if no valid P/S is found, unknown next hop) or forwarded to one or multiple P/S (depending if they are multicast flows).
- **Cherish/Urgency Multiplexer**: This module process the packets received from the distinct P/S and process them strictly depending on

their cherish/urgency values. First, when receiving packets, it decides whether to drop them depending on the cherish value of the flow and the number of packets waiting to be sent on that port. Then, when the port is ready to send new data, it selects the next packet to be sent, given the urgency of the supported flows. Since the urgency value can be degraded for some packets in order to adjust the average delay of the flow, it is important to ensure that the order between PDUs of the same flow is maintained.

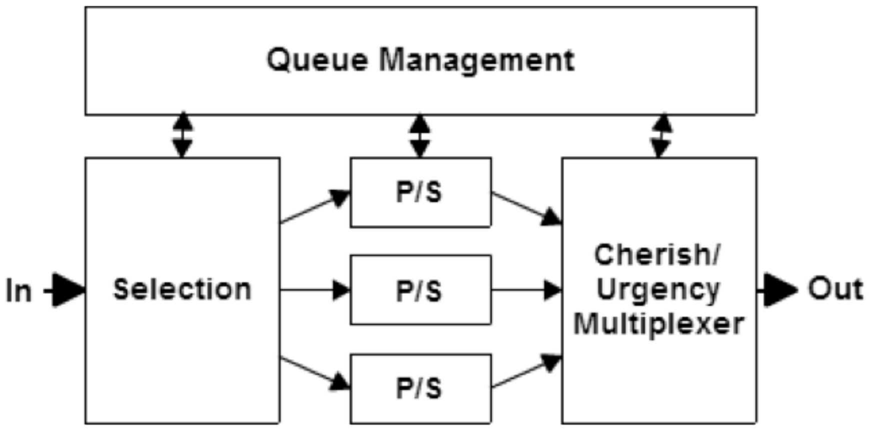


Figure 17. Simplified view of ΔQ sub-modules.

In more detail, the following features become key to the proper ΔQ operation:

- **At the Policer/Shaper (P/S) before the multiplexer**:
    - *Flow bandwidth/ratio control*. To impose limitations when serving urgent flows in order to avoid starvation of low urgency flows under high congestion. Moreover, bandwidth limitations per QoS are required between domains in order to avoid that flows originated in one domain saturate the second domain with urgent flows.
    - *Packet spacing*. To perform smoothing of data bursts to more constant rates, while adding small variable inter-PDU delays helping to avoid starvation of low urgency flows/QoS.
    - *Flow degradation*. To degrade the QoS class of PDUs belonging to flows consuming more resources than those initially agreed. This measure complements the ratio control, degrading the service that PDUs experience when approaching the situation that will cause them to be dropped
- **At the multiplexer**:

  ◦ *Drop PDUs of uncherished flows upon congestion*. Depending on the output port buffer occupation, PDUs of uncherished flows are dropped leaving more space to PDUs cherished flows. As the buffer occupation increases, the range of cherished classes with losses also increments, allowing for the most cherished flows to avoid losses until the very end.

  ◦ *Serve PDUs depending on their urgency*. PDUs of more urgent flows are always served before those with lower urgency value, thus ensuring lower delay on urgent PDUs.

2.1.3. Adaptation of the ΔQ approach to RINA

From the RINA IPCP architecture and operation point of view, the aforementioned ΔQ features can be viewed as:

- Limit Flow rate on input port per flow, QoS or Cherish/Urgency class.
- Space PDUs on input ports.
- Degrade flow, QoS or Cherish/Urgency class of PDUs upon resource overuse.
- Cherish/Urgency Multiplexing on output ports.

While ΔQ works "per flow", it leaves the definition of flow quite open (same src/dst + qos, same dst + qos, same dst port + qos, etc.). Given that, while ΔQ perfectly fits a connection-oriented scenario treating N-level flows individually, we plan to focus on a connectionless approach, where "flows" are defined by a certain IPCP output port and cherish/urgency values (note that multiple QoS Cubes can have the same cherish/urgency values).

ΔQ scheduling will be performed between 3 RMT policies: Scheduling Policy, MaxQ Policy, Q Monitor Policy.

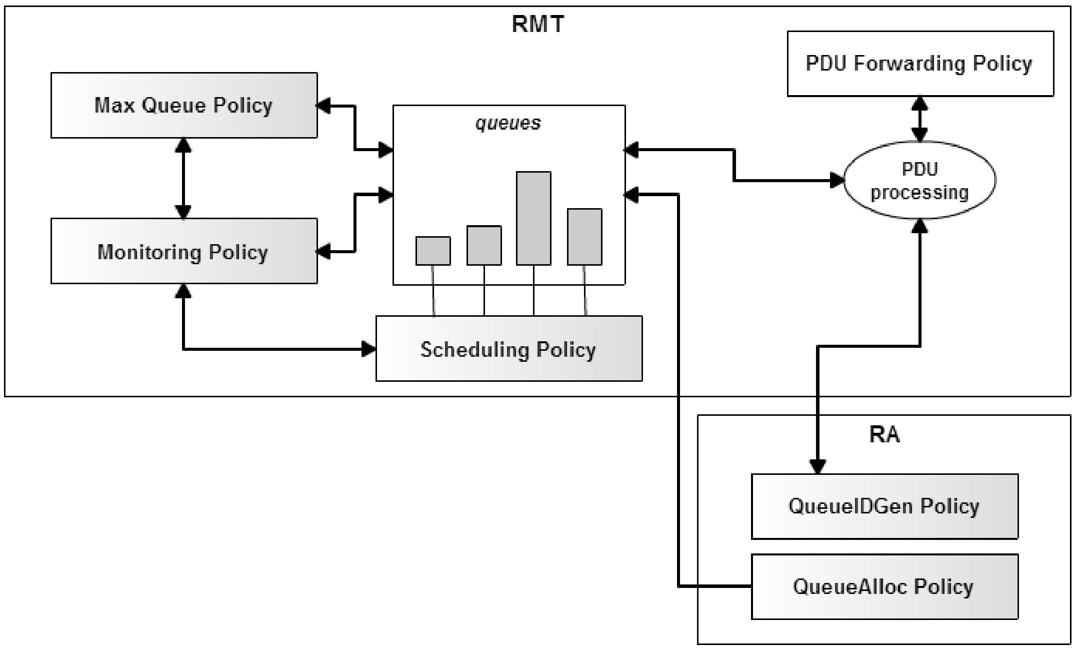


Figure 18. RINA policies related to the ΔQ adaptation

Scheduling Policy

This policy is in charge of deciding what Queue must be served next, and is called any time a PDU is successfully inserted into a queue (not dropped) or the port/RMT core is ready to process a new PDU and some are waiting in its queues. In ΔQ scheduling in RINA, this policy queries the Monitor policy for the next queue to serve and serves the next PDU of it. In case of input ports, if the monitor responds to the query of next queue to serve with a NULL Queue pointer, the scheduling policy requests the next serving time computed for that given port and schedules a new call at that time.

Max Q Policy

This policy is responsible for deciding if the last inserted PDU in a queue has to be dropped or not. In ΔQ scheduling in RINA, this policy queries the Monitor policy for PDU dropping probability of that queue, and decides, according to this probability, to drop the PDU randomly.

Q Monitor Policy

The Monitor policy of ΔQ is the core of the ΔQ scheduling in RINA. This policy is responsible for monitoring bursts of bandwidth usage at each input port queue in order to limit the bandwidth usage of input flows, thus spacing PDUs in this way. It also monitors output ports, computing how PDUs are sent based on their urgency and arrival time and decides when to degrade the urgency of individual PDUs in the queues if exceeding the agreed data rate.

The Configuration parameters for this policy are:

- **bool** *LimitIn*. If true, enable BW control on input.
- **bool** *space*. If true, enable spacing on input.
- **bool** *degradation*. If true, enable degradation of urgency on output.
- **map**<**Queue**, int >* *queueUrgency*. Default urgency/priority of an output queue.
- **map**<**Queue**, vector< pair< int, double[0..1] > > >* *dropOutProb*. Probability of degrading the urgency in an output queue, given the total number of PDUs waiting to be served in the output port.

If *LimitIn* is enabled:

- map<Queue*, vector< pair< int, double[0..1] > > > *inRates*. Rate in which an input Queue is served, given the number of PDUs waiting to be served at future time.
- map<Queue*, vector< pair< int, double[0..1] > > > *dropInProb*. Probability of dropping messages from an input queue, given the number of PDUs waiting to be served at future time.

If *space* is enabled:

- map<Queue*, vector< pair< int, pair< double, double > > > > *inSpaces*. Range of delays to space PDUs on an input queue depending on the number of PDUs waiting to be served at future time.

If *degradation* is enabled:

- map<Queue*, vector< pair< int, double[0..1] > > > *outRates*. Rate in which an output Queue is served, given the number of PDUs waiting to be served at future time.
- map<Queue*, vector< pair< int, double[0..1] > > > *degOutProb*. Probability of degrading the urgency in an output queue, given the number of PDUs waiting to be served at future time.

In the following lines, the targeted functionality (when all procedures are enabled) is briefly described:

- When a PDU arrives at an input queue: Check the number of PDUs with computed serving time after the current for that queue and get the current rate from inRates and range of delay from inSpaces. Compute the serving time for the inserted PDU given the last computed time and the rate, and the expected serving time given the range of delay and the computed for the last PDU. Also remove all the computed serving times for that queue with already passed.
- When a PDU is dropped from an input queue: Remove the last computed service time and expected serving time for that queue.
- When a PDU departs from an input queue: Remove the first computed expected service time for that queue.
- When queried for the drop probability of an output queue at input: Check the number of PDUs with computed service time after current time for that queue and get the drop probability from dropInProb. Return that probability.
- When queried for the next queue to serve, at input port: Search the queue with the lowest expected service time. If that time is lower or equal than the current time, return it, otherwise return NULL.
- When queried for the next serving time: Search the lowest expected serving time and return it.
- When a PDU arrives at an output queue: Check the number of PDUs with computed service time after the current time for that queue and get the probability of degradation and current rate from degOutProb and outRates. Get the default urgency for the queue from queueUrgency, and degrade it (urgency -1) randomly, given the probability of degradation obtained from degOutProb. Put a pointer to the queue into a priority queue of Queue pointers, with the urgency as priority value. Compute the serving time for the inserted PDU given the last computed time and the rate.
- When a PDU dropped from an output queue: Remove the last queue pointer inserted for that queue. Remove the last computed service time for that queue.
- When queried for the drop probability of an output queue at output: Check the number of PDUs with computed service time after the current

for that queue and get the drop probability from dropOutProb. Return that probability.

- When queried for the next queue to serve, at output port: Extract the first Queue pointer from the priority queue of Queue pointers and return it.

2.1.4. Delay/Loss Scheduling - Cherish/Urgency Multiplexing

As illustrated in the previous section, the complete ΔQ implementation entails substantial complexity in the RINA IPCP policies. While the complete ΔQ implementation in RINA is our ultimate goal at the end of the PRISTINE second iteration, in the first iteration we have implemented and evaluated using the RINA simulator simpler policies for basic cherish/urgency multiplexing within RINA, which can be seen as a first step toward the complete ΔQ implementation goal.

Delay/Loss queuing is a simple scheduling algorithm that mixes priority queue scheduling with distinct thresholds in order to make a strict differentiation of services based on urgency and cherish of flows. In Delay/Loss, cherish/urgency class is assigned to each output queue given the QoS Cube assigned to the PDUs of that queue. For each cherish/urgency class, the scheduling algorithm can ensure a maximum and average delay and losses given a normal usage of the network (at most 100% usage plus bursts, not only urgent/cherished flows, etc.).

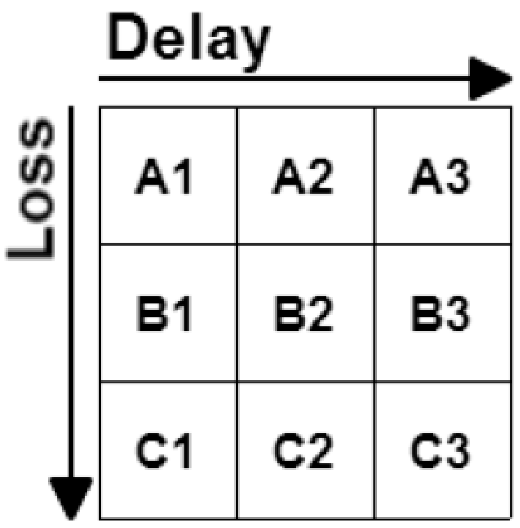


Figure 19. Cherish/Urgency classes, 3x3 matrix

Delay/Loss scheduling does a similar Cherish/Urgency Multiplexing on output ports as in the complete ΔQ, but removes the bandwidth control and

PDU spacing on input ports, as well as the PDU degradation and dropping ranges on output ports, hence leading to simplified policies.

While Delay/Loss is a scheduling algorithm used to differentiate services based on QoS Cubes, it does not impose any requirements on the number of queues used or how the flows from where these queues are populated. It only requires for each queue to have PDUs of QoS Cubes defining the same cherish/urgency values. Given that, we can use Delay/Loss scheduling with multiple queuing schemes, like one singe queue per flow, or one queue per QoS cube, a queue per cherish/urgency tuple, etc.. This makes possible to use the Delay/Loss algorithm in conjunction with other scheduling algorithms, like a fair queuing scheduling between flows with the same cherish level (with small modifications on the policies).

While Delay/Loss provides clear differentiation between flows given their urgency and cherish values, the strict priority and fixed thresholds easily leads to resources starvation for the flows with lower priorities. In order to solve that, while still avoiding the complexity of the full ΔQ implementation, we have also evaluated an improved version of the Delay/Loss scheduling that we have called Enhanced Delay/Loss.

Unlike the basic Delay/Loss, in Enhanced Delay/Loss, when deciding if an arriving PDU has to be dropped, it not only considers a maximum occupation threshold given the total occupation of all queues connected to the output port, but also considers an small threshold for which it will randomly drop the PDU with certain probability, both also defined by the cherish level of the queue. In addition, each urgency level has a skip probability. Given that probability, when deciding the next queue to serve, it is possible to skip queues with higher urgency, thus giving room to serve PDUs of queues with lower urgency.

Draft policies

Delay/Loss scheduling requires the joint collaboration of the 3 RMT policies: *Scheduling Policy*, *MaxQ Policy*, *Monitor Policy*.

Scheduling policy

This policy is in charge of deciding what Queue serve next and is called any time a PDU is inserted into a queue (and not dropped). In Delay/Loss and

Enhanced Delay/Loss scheduling in RINA, this policy queries the Monitor policy for the next queue to serve and serves the next PDU waiting there.

Max Q policy

This policy is in charge of deciding if the last inserted PDU in a queue has to be dropped or not. In Delay/Loss and Enhanced Delay/Loss scheduling on RINA, this policy queries the Monitor policy for the tuple <occupation, threshold, dropProb, absThreshold> for a given queue, and then it decides if that PDU must be dropped. The pseudo-code for dropping decision is presented as follows:

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

Monitor policy

The Monitor policy is the core of the Delay/Loss and Enhanced Delay/ Loss scheduling in RINA. This policy is responsible for monitoring the usage at each output port and for computing how PDUs are sent based on their urgency and arrival time. PRISTINE has worked in two variants of this policy: **delay/loss monitor** and **enhanced delay/loss monitor**. A brief sketch of its functionality is presented as follows:

- When a PDU arrives at an input queue: insert the queue into a queue of Queue pointers.
- When a PDU is dropped from an input queue: remove the last Queue pointer.
- When a PDU departs from an input queue: do nothing.
- When queried for the drop probability of an output queue at an input port: return the tuple <queue.length, queue.threshold, 1, queue.threshold>.
- When queried for the next queue to serve at an input port: extract the first Queue pointer from the queue of Queue pointers and return it.
- When a PDU arrives at an output queue: get the urgency for the queue. Put a pointer to the queue into a priority queue of Queue pointers, with the urgency as priority value. Increment the PDU count for that port.

- When a PDU is dropped from an output queue: remove the last queue pointer inserted for that queue. Decrement the PDU count for the port.
- When PDU departs from an output queue: decrement the PDU count for the port.
- When queried for the drop probability of an output queue at output.
    - With the *Delay/Loss scheduling*: Check the number of PDUs waiting on the port of the given queue. Get the threshold for that queue. Return the tuple <waiting pdus, threshold, 1, threshold>.
    - With the *Enhanced Delay/Loss scheduling*: Check the number of PDUs waiting on the port of the given queue. Get the threshold, drop probability and absolute threshold for that queue. Return the tuple <waiting pdus, threshold, drop probability, absolute threshold>.
- When queried for the next queue to serve, at output port.
    - With the *Delay/Loss scheduling*: Extract the first Queue pointer from the priority queue of Queue pointers and return it.
    - With the *Enhanced Delay/Loss scheduling*: Search the first urgency with PDUs to serve. Given the probability to skip that urgency, decide to serve it or skip it. If skipped, search for the next urgency with PDUs to serve, and repeat, until selecting the next urgency to serve. Finally, extract the first Queue pointer from the priority queue of Queue pointers with the selected priority and return it.

2.1.5. Simulation results

Some simulations have been obtained using the RINA sim in order to evaluate the performance of the Delay/Loss and Enhanced Delay/Loss scheduling policies. The obtained results are presented throughout this section.

Delay/Loss vs Best Effort

The first tests that we have conducted are intended to compare the basic *Delay/Loss scheduling*, with only two levels of cherish and urgency, against a benchmark Best Effort scheduling, with only a single shared queue per output port. For these tests, illustrated by the Figure below, our scenario was a simple network with one router, one server and 4 PCs that communicate with the server. In this case, the PCs run 4 applications each, each application sending a flow of data consuming nearly 25% of the bandwidth from PC to router, each with distinct QoS requirements (Urgent and cherished<A1>, Urgent<B1>, Cherished<A2> and Best Effort<B2>). From router to host we provision 4 times the bandwidth that we have from the PCs to the router, so there are 2 bottlenecks in the network (PCs#router and router#server). The generated data flows are assumed to be bursty, following an ON-OFF traffic profile. ON traffic periods are composed of 1 to 10 PDUs (uniformly distributed) having each one a size of 1024 bits (+/- 400 bits, uniformly distributed) + headers. OFF (i.e., idle) traffic periods are also assumed to have duration equal to the duration of the complete previous burst. In any case, we would like to mention here that these assumed traffic profiles allow us to perform an initial evaluation of our QoS scheduling proposals. For the final simulation results, however, we plan to generate more realistic traffic profiles, tightly following the traffic behaviour of real data applications (HTTP, VoIP, etc.).

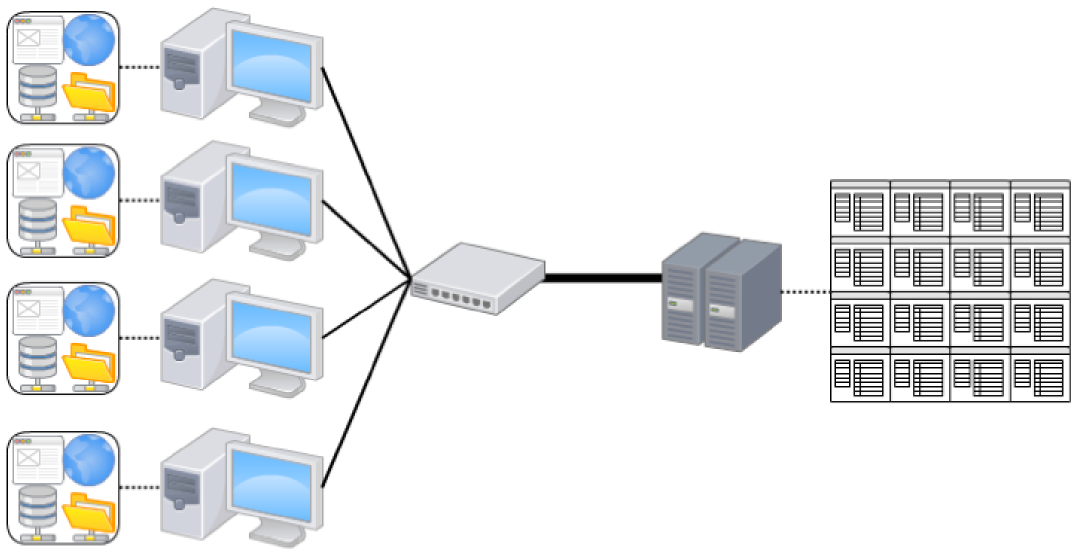


Figure 20. Scenario for Delay/Loss 2x2 testing.

The results of the tests are presented in the following figure. While in best effort we encounter high losses and delay (Round Trip Time, RTT, delays are measured, assuming 0 ms link propagation times, thus only accounting for buffering delays) for all flows, no matter what their QoS requirements are (i.e., no differentiation between them is made), in Delay/Loss we can appreciate that differentiated treatment is provided to the flows belonging to the different QoS classes, effectively providing their delay and loss demands. As seen, losses are avoided for the cherished flows (PDU loss probability 0%), compared to the 1.77% PDU loss probability observed in the best effort scenario. Of course, ensuring such a low PDU loss probability for the cherished flows comes at expenses of increased losses experienced by the uncherished flows, rising up to around 3.5%. Moreover, urgent flows also experience substantially lower delays (approximately 10 times lower average/maximum PDU delay) when compared the not urgent ones.

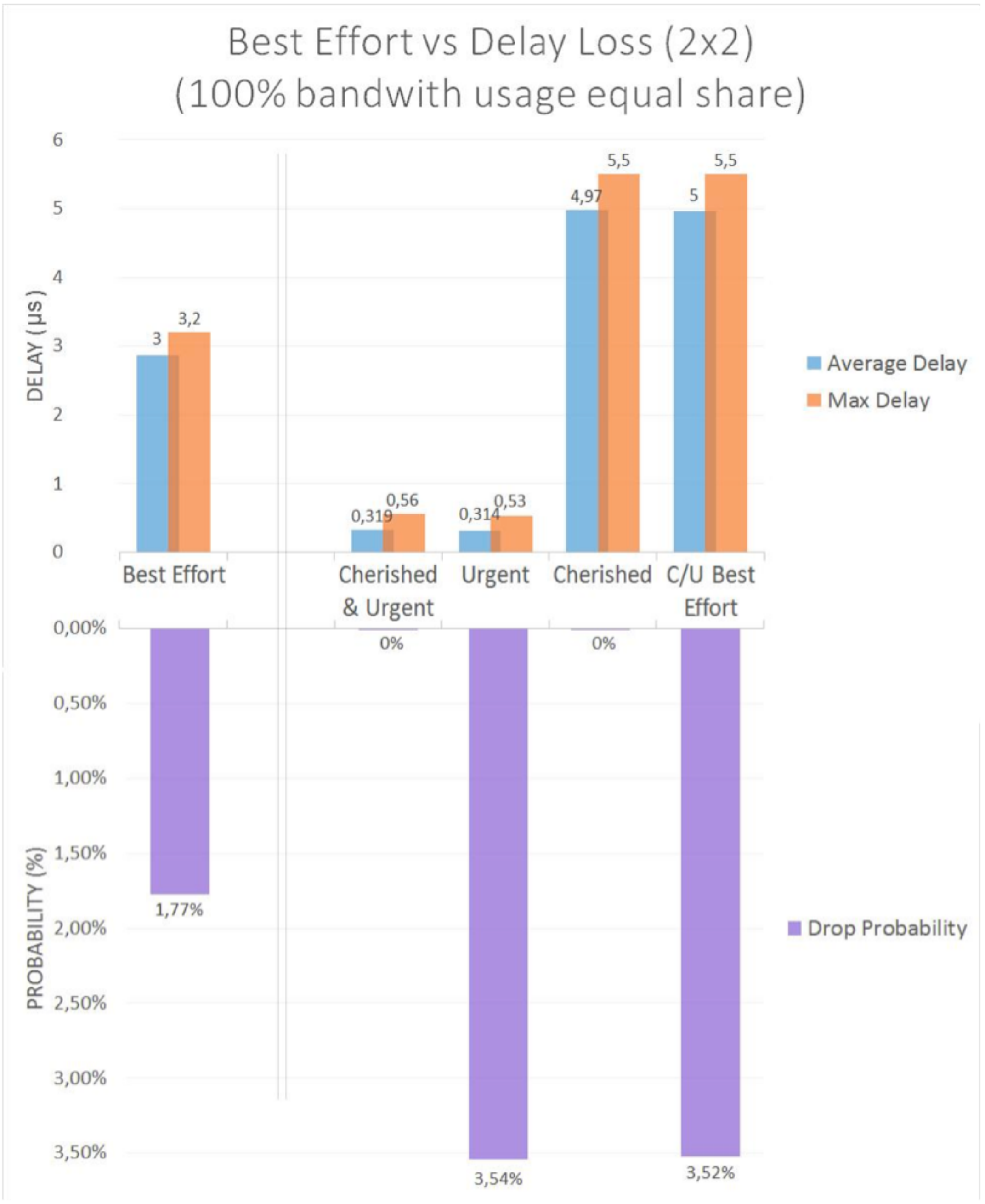


Figure 21. Comparative Delay and losses. Best Effort vs Delay/Loss 2x2 QoS classes

Delay/Loss vs Enhanced Delay/Loss

After having seen the benefits of Delay/Loss with respect to Best Effort, more complex tests with bigger cherish/urgency matrices (3x3) show us that, while the Delay/Loss scheduling maintains the strict order when serving cherish/urgency classes (+ Urgency = - Delay, + Cherish = - Losses), it negatively affects the least cherished/urgent classes. Looking at the following figure, we can see that the distinct QoS are served strictly following their requirements (Cherish (A) has less losses than Low Cherished traffic (B) and that less than Uncherished ©, and Urgent (1) has less delay than Low Urgency (2) and that less than Not Urgent (3)), there is really no feasible way of adjust how the distinct QoS are served, ending with really similar delays / losses in all the classes except in the Non Urgent / Uncherished ones.

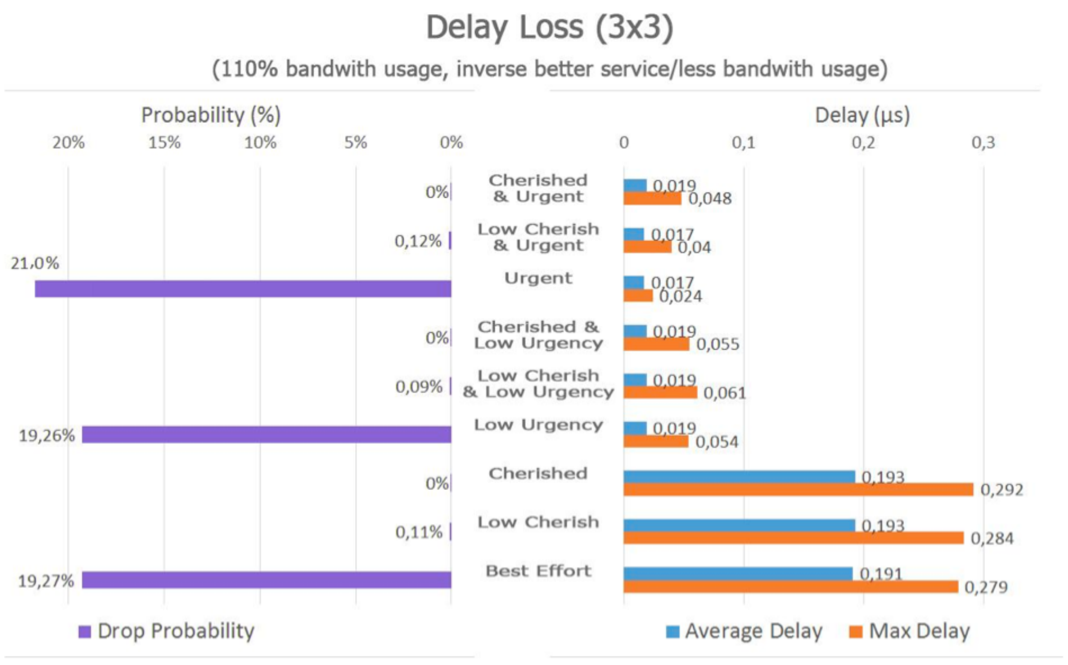


Figure 22. Comparative Delay and losses. Delay/Loss 3x3 QoS classes

These large differences between QoS cubes using the Delay/Loss scheduling were the motivation behind the proposal of the Enhanced Delay/Loss one. To highlight the benefits of the latter, we compare their behavior in a similar scenario as before. In this case, our scenario was built using 2 PCs interconnected, each with 45 pairs of applications communicating between them. Among these 45 pairs of applications, 9 groups of 5 applications were assumed, each group of applications requesting flows of a certain QoS class (from the 9 available in the 3x3 QoS class matrix). In order to simulate a realistic scenario, bandwidth usage between the distinct QoS was divided in a way that ~5% of the traffic was urgent, ~40% had low urgency and the remaining ~55% was non-urgent.

We have tested the same scenario with Enhanced Delay/Loss. In this case, given the extra possibilities of Enhanced Delay/Loss, we configured it with the idea of, in terms of urgency, provide to urgent classes the same minimum delay as before, but distribute the delay between less urgent flows in a fairest way. The same was done for the experienced PDU losses, remaining the highest cherished QoS cubes without losses whenever possible, and then distribute them fairly between the least cherished QoS. The next figure show the results obtained for the Enhanced Delay/Loss scheduling. In the plotted bar graphs, we can see that, in addition to maintain the strict distinction between how the QoS are served depending of their cherish/urgency values, the results obtained show a scenario where the distinct QoS becomes clearly distinguished, but the differences between Uncherished / Non urgent and the rest of classes are not so abrupt.

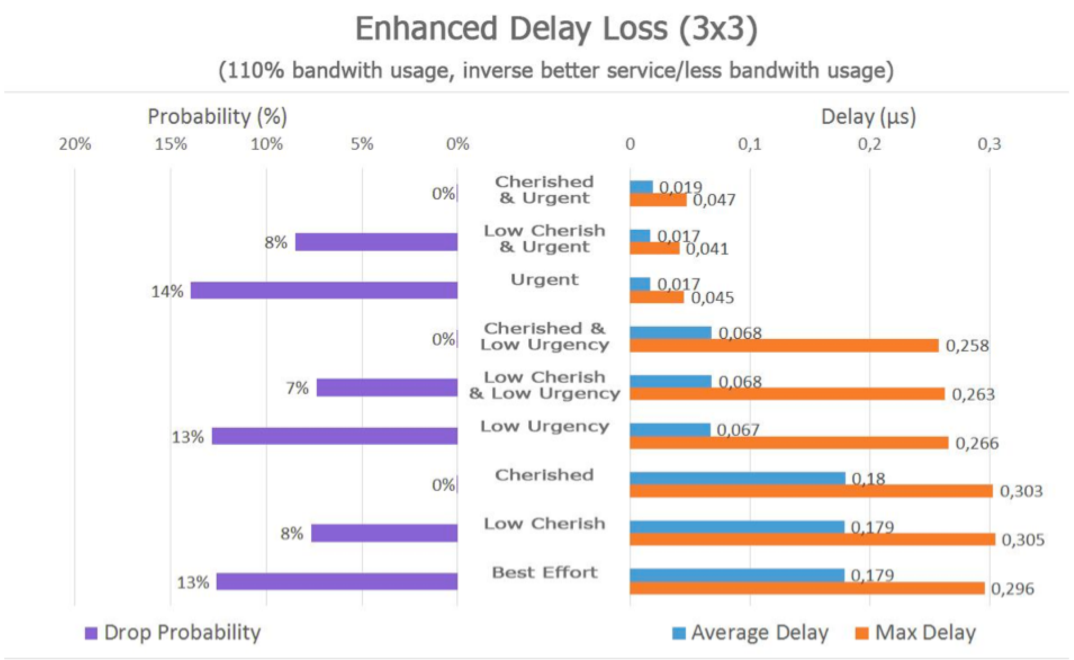
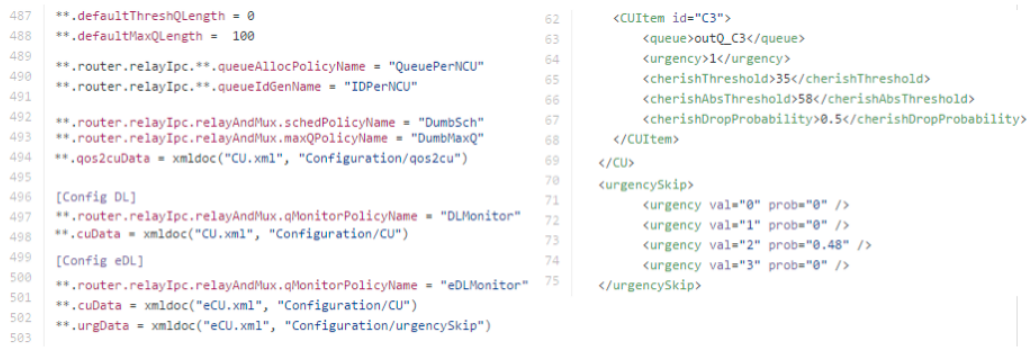


Figure 23. Comparative Delay and losses. Enhanced Delay/Loss 3x3 QoS classes



```
**.defaultThreshQLength = 0
**.defaultMaxQLength =  100

**.router.relayIpc.**.queueAllocPolicyName = "QueuePerNCU"
**.router.relayIpc.**.queueIdGenName = "IDPerNCU"

**.router.relayIpc.relayAndMux.schedPolicyName = "DumbSch"
**.router.relayIpc.relayAndMux.maxQPolicyName = "DumbMaxQ"
**.qos2cuData = xmldoc("CU.xml", "Configuration/qos2cu")

[Config DL]
**.router.relayIpc.relayAndMux.qMonitorPolicyName = "DLMonitor"
**.cuData = xmldoc("CU.xml", "Configuration/CU")
[Config eDL]
**.router.relayIpc.relayAndMux.qMonitorPolicyName = "eDLMonitor"
**.cuData = xmldoc("eCU.xml", "Configuration/CU")
**.urgData = xmldoc("eCU.xml", "Configuration/urgencySkip")
```

```
    <CUItem id="C3">
        <queue>outQ_C3</queue>
        <urgency>1</urgency>
        <cherishThreshold>35</cherishThreshold>
        <cherishAbsThreshold>58</cherishAbsThreshold>
        <cherishDropProbability>0.5</cherishDropProbability>
    </CUItem>
</CU>
<urgencySkip>
        <urgency val="0" prob="0" />
        <urgency val="1" prob="0" />
        <urgency val="2" prob="0.48" />
        <urgency val="3" prob="0" />
</urgencySkip>
```

Figure 24. Configuration example for RINA sim. Left: ini file for Delay/ Loss and Enhanced Delay/Loss. Right: xml for Enhanced Delay/Loss

2.1.6. Next Steps

Delay/Loss policies have shown good results in controlled environments like the ones simulated with RINA sim. Thus, our next step will be to prototype them using the PRISTINE RINA SDK. In addition, we are going to continue the work with the more complex scheduling that ΔQ. For that, we are going to do the full implementation in the RINA sim in order to finish adjusting the distinct policies and test its behavior. At the end, we expect to implement a complete and configurable set of ΔQ scheduling policies within the RINA SDK.

2.2. QoS-aware Multipath Routing

2.2.1. Multipath Routing Overview

Multipath routing refers to routing strategies in which traffic is delivered through multiple paths. Sender or intermediate nodes have several next-hops for a given destination and must choose the next-hop for a given packet. Multipath routing is used for performance and traffic engineering purposes such as load balancing or congestion avoidance among others. In the following we outline common multipath routing strategies.

Equal Cost Multi-Path (ECMP)

Equal-cost multi-path routing (ECMP) is a routing strategy that distributes traffic among equal-cost routes to the same destination, typically using Round Robin or random approaches. ECMP is a per-hop decision that is limited to a single router. It is used for load balancing, also offering increases in bandwidth.

ECMP presents the drawback of variable latency among the paths causing packets to arrive out of order, increasing delivery latency and buffering requirements. This problem arises when packets in the flow are split among multiple paths. Therefore, to avoid this situation the natural solution is to deliver packets belonging to the same flow through the same path. [RFC2991] describes some solutions for a router to select the same path (next-hop) per flow, namely: Modulo-N Hash, Hash-Threshold and Highest Random Weight. These solutions rely on some form of hash function.

ECMP always split traffic evenly among the available paths, which is not the optimal solution for variable and non-balanced scenarios. Weighted ECMP aims to address this problem.

Weighted ECMP

Weighted ECMP is a form of ECMP which distributes traffic among paths based on a set of pre-determined ratios. Heuristics are used to find optimal traffic distribution (link weights) based on source routing approaches (explained below).

However, [Chiesa] demonstrates that optimizing link weight configuration or even achieving a good approximation to the optimum is an infeasible task, which is a huge drawback for weighted ECMP.

Source routing

Source routing is a routing strategy by which the sender partially or completely specifies packets' route.

In strict source routing, the sender specifies the exact route the packet must take, but this is never used in practice. Loose Source Record Route (LSRR) is more commonly used, in which the sender gives one or more hops that the packet must go through.

Source routing allows a network manager to perform the routing functionality and decide the routes the traffic traverses. Also, alternate links can be used upon changing conditions, for example to avoid congested links.

Policy-based routing

Policy-based routing (PBR) is a routing strategy that makes routing decisions based on pre-defined policies. This allows forwarding packets based on varied criteria such as the source address instead of the destination address, the size of the packet, the protocol of the payload, or other information available in a packet header or payload.

Dynamic Adaptive Routing

Adaptive routing is a common characteristic of routing protocols such as RIP or OSPF, which refers to the ability to alter the path that the route takes through the system in response to a change in conditions. For example, if a certain node crashes, another feasible path (if any) is discovered and chosen by the protocol to reach the affected destinations. The opposite of adaptive routing is static routing, in which the paths are fixed and failures in them leads to connection breaks.

Dynamic Adaptive Routing aims to react upon dynamic conditions such as link congestion level, i.e. choosing the best paths based on dynamic conditions that damage QoS. This kind of multipath routing is so far not well studied, being only addressed in highly variable networks such as sensor networks or heterogeneous networks (HETNETs). Besides, its implementation poses many challenges mainly due to the difficulty of cross-layer optimization between the transport and network layer.

2.2.2. QoS-aware multipath routing

A general drawback presented by the existing multipath solutions is that application requirements are not considered. This leads to non-optimal multipath solutions since some applications are delay-, jitter- and loss-sensitive, and therefore not suitable for multipath approaches, while others are not sensible to these parameters and may benefit greatly of the extra bandwidth achieved by using multiple available paths.

As an example, video conference applications are very sensible to delay and jitter, and therefore require a single path approach to avoid packet reordering and buffering delays. In contrast, file transfer applications just need bandwidth to deliver the content as fast as possible, and do not care about delay and jitter.

Therefore, considering the application QoS requirements when taking the decision of whether to do multipath or not, or deciding how many paths to split the traffic among, can lead to consistent benefits in terms of performance and quality perceived by the end-user.

Multipath level

The next figure outlines a tentative arrangement of different application types according to their “optimal” multipath level, i.e. the number of available paths they should use to deliver optimal performance. This is just an exercise to show the importance of the application requirements in multipath routing and the figure does not show experimental results of any kind.

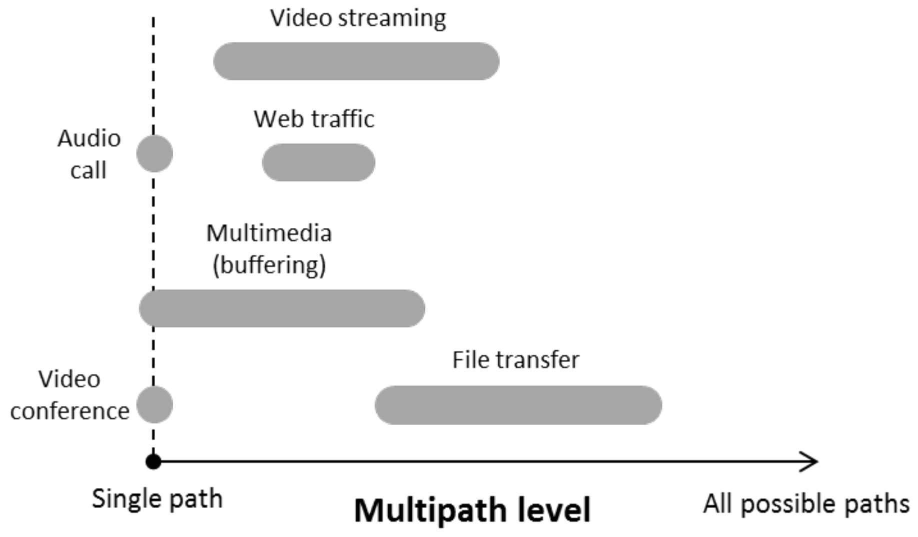


Figure 25. Multipath level

For delay- and jitter-sensitive applications such as video conference and audio calls, the most appropriate approach **might be to use a single path. However, in case the delay variance and re-ordering delay overhead keep under the delay QoS requirements, multiple paths can be also used without affecting the required performance.**

Bandwidth-demanding applications such as buffered multimedia or video streaming may benefit of the bandwidth obtained by means of multiple paths, but always keeping an eye on the multipath drawbacks on QoS parameters such as delay, jitter and packet losses, which may require the utilization of less (or even a single) paths. Web traffic can benefit largely from multipath, since web browsers gather the different web objects using separate connections and multiple paths can be leveraged. Some web content may require high bandwidth, however the bandwidth requirements are not that high as for multimedia applications and the maximum multipath level can be maintained below theirs. Finally, bandwidth demanding applications which have not delay, jitter or packet loss requirements may benefit greatly from a high multipath level. For example file transfer applications. They may use the maximum bandwidth provided by the different available paths to transmit the content as fast as possible. However, in some cases using all the available paths may have additional drawbacks. Especially when the number of available paths is very high, and the forwarding through all of them shall be avoided.

When to do multipath?

Current multipath routing solutions are applied in a pre-defined way. Network nodes already have a pre-defined multipath level to follow, and the question of whether to perform multipath or not is never asked nor answered. For example, network nodes may have an ECMP routing strategy installed, and they will always spread traffic among paths.

Weighted ECMP allows some degree of configuration by means of dynamically varying the link weights, but this is more oriented to dynamic load balancing rather than taking a true decision on whether to perform multipath or not. A true dynamic weighted ECMP can serve as the basis to achieve dynamic configurations of different multipath levels. For example, a link weight of zero may indicate that traffic is not to be forwarded through it, and in case this link is allowed for multipath, its weight can be changed. However, this approach also carries the drawback of not considering the kind of traffic that is being forwarded, since the link weights apply for all the traffic that reaches and leaves a certain node.

Therefore, to take dynamic decisions on whether to perform multipath and what multipath level applies, the consideration of the application and differentiation of traffic type is mandatory. In the following we elaborate how such an approach can be implemented using RINA.

Multipath level

We can define multipath level as a measure that determines the type of multipath routing strategies associated with the forwarding of a certain traffic type through different possible paths, which allows taking the decision on what next hop to choose for each packet.

This measure can be unidimensional or multidimensional depending on the degree of information that it conveys to the policies that take multipath routing decisions. For example, a unidimensional measure from 0 to 1 may indicate the desired multipath level from a single path (0) to all possible paths (1), or something in between. Another possibility is to use a bi-dimensional measure indicating the previous metric together with the typical deviation of the traffic load to be forwarded through different paths, so that the major traffic load is concentrated in a reduced number of paths.

Other aspect that may be considered for the multipath level is whether multipath is performed as long as multiple paths are available, or if multipath takes place upon certain conditions (e.g. congestion, as MPTCP).

In summary, for specifying the multipath level we can use any multi-dimensional measure that we need to convey the needed information to the routing policies, as long as those policies understand the information that this measure includes.

Multipath-based Resource Allocation

Apart from the multipath level, resource allocation is a key aspect for the routing decisions. In fact, multipath routing can be understood as a way of resource allocation, since routing traffic through a certain path means to utilize or allocate that network resources for the routed traffic.

The driving paradigm of resource allocation is to optimally utilize the resources available to it (maintaining the appropriate safety levels), while responding to requests for service and satisfying those that it can while still staying within its policy bounds. What resources to allocate to what entity is a matter of the resource allocation approach. QoS-aware multipath routing in this case can be seen as a way of providing further information to the resource allocation mechanisms to share the available network resources more efficiently. I.e. the resource allocation mechanisms will know when to dedicate resources of the same link to single path oriented traffic and when to share link capacity among multi-path oriented traffic. Therefore, the result of these resource allocation mechanisms is to influence the forwarding decision, which leads to a multipath-based resource allocation approach.

2.2.3. Multipath levels in RINA

In RINA, when a flow request is issued, the QoS requirements are passed to the DIF as part of the flow allocation requests. The QoS requirements are then compared with the QoS cubes provided by the DIF (each QoS cube can be thought of a set of policies that guarantee a specific level of service), and, in case the request can be honored by the DIF, the most appropriate

QoS cube is selected. Then, the different policies forward the traffic to comply with those QoS specifications. Therefore, when an application requests a flow allocation, its requirements are indicated by means of the QoS parameters. In order to determine the multipath level associated with that application, the multipath level is derived from the QoS parameters indicated by the application.

Individual applications cannot request what is the multipath level they want for their flow for two reasons. Applications care about the **outcomes** of the service the DIF provides, not about how the service is implemented: if the flow is transported over a single or multiple paths over the DIF is a detail of the service implementation; as long as the DIF guarantees the flow characteristics requested by the application, the application does not care. In other words, the application tells the DIF **what outcomes it wants**, not **how to achieve them**. The second reason is that the DIF is a black box; the application has no visibility inside, and therefore cannot know the paths the DIF can use.

Derivation of the multipath level from the QoS requirements

The multipath level associated with a certain flow can be derived by means of the QoS requirements associated to the flow (e.g. jitter, delay, bandwidth, etc.). In fact, this may imply not considering any multipath level at all, since the multipath decision policies can take directly the needed QoS parameters to take the decision. We use the multipath level concept here for the sake of clarity.

QoS parameters [D22] that can be considered to determine the multipath level are:

- Average bandwidth: Unsigned Integer - measured at the application in bits/sec
- Average SDU bandwidth: Unsigned Integer - measured in SDUs/sec
- Peak bandwidth-duration: Unsigned Integer - measured in bits/sec
- Peak SDU bandwidth-duration: Unsigned Integer - measured in SDUs/sec

Higher bandwidth requirements demand the utilization of additional paths to fulfill the requirements.

- Burst period: Unsigned Integer - measured in seconds
- Burst duration: Unsigned Integer - measured in fraction of Burst Period

Bursts may demand the utilization of single paths. So burst periods and durations must be considered to allocate those bursts accordingly among the different possible paths.

- MaxDelay: Unsigned Integer - in secs

The delay may constrain the multipath decision, as the delay varies among paths. Under delay constraints, some paths might be out of the scope, and therefore the multipath level reduced.

- Jitter: Unsigned Integer - in secs

Tight jitter constraints demand low multipath level (including single path). In case of loose jitter constraints, additional paths may be used freely.

2.2.4. QoS-Aware Multipath Routing for RINA

This section describes how the presented QoS-aware multipath routing and dynamic adaptive routing concepts can be implemented in RINA, studying what RINA components and policies are involved in the multipath routing approach.

The main requirement for forwarding is to take fast decisions (since it is a function performed for each PDU). Thus the amount of required computation to forward a packet should be small. For example, when following a round-robin approach, the computation might consist of incrementing a next-hop index, keeping the delay overhead at a minimum. This involves low computing overhead, i.e. the amount of computing involved in the forwarding strategy shall be small. The computation of paths, next hop to use, etc. shall be carried out fast and in a non-redundant manner. To that end, pre-computed data will be stored to be accessed by routing algorithms, thus achieving both optimal routing and fast decisions.

Figure 26 illustrates, from a high-level point of view, the scope of the multipath approach within the involved RINA components, namely the Relaying and Multiplexing Task (RMT), Resource Allocator (RA) and Routing.

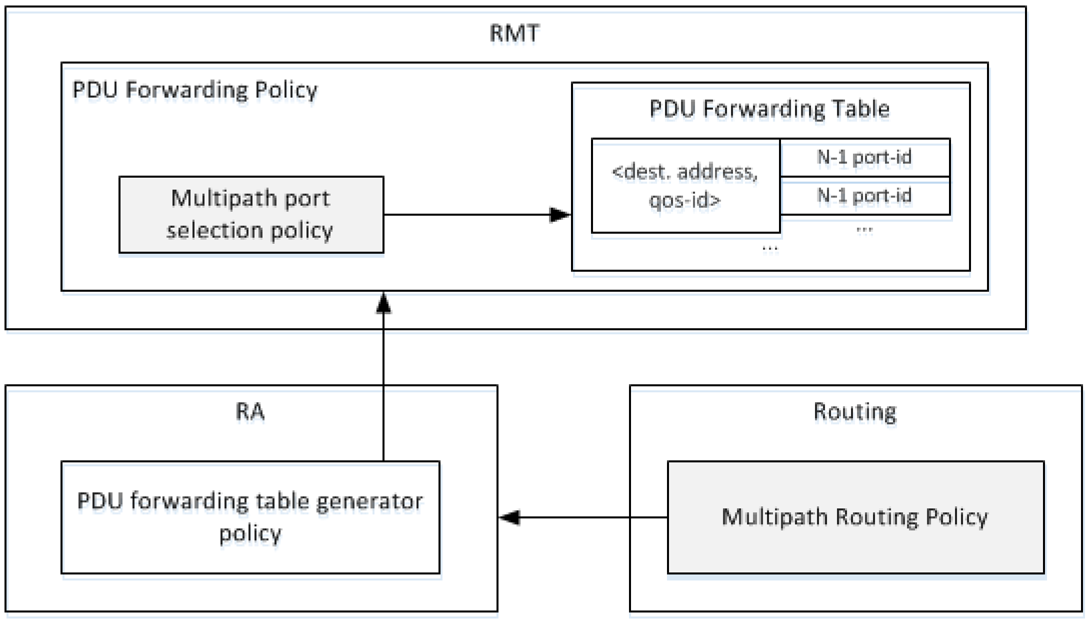


Figure 26. Multipath routing design

Routing

Routing exchanges connectivity and other state information with other IPC processes of the DIF and holds the routing table generation algorithms. Routing performs the analysis of the information maintained by the RIB to provide connectivity input for the creation of the forwarding table. Routing executes when it is needed (periodically, based on routing updates from neighbors, etc). It computes routes to next hop addresses and provides this information to the Resource Allocator.

Multipath routing policy

Since routing is computationally expensive, generally is not worth it to **execute routing algorithms to take forwarding decisions** on a PDU-basis. Instead, routing algorithms are executed with more time granularity, being the results stored in the PDU forwarding table to be consulted in a fast manner to reduce the **forwarding decision** time.

In multipath routing, the output of the Routing component must comprise the different **next hop addresses of the existing available paths** to reach a certain destination, including figures of merit associated with each **next hop** to be used in the PDU forwarding decision process. Always having in mind that this decision should be as fast as possible, the resource allocator must provide the necessary pre-computed information to optimize the **forwarding decision time**. This is a matter of synchronization between

Routing and the PDU forwarding policy to reach the optimization goal in a coordinated way. For example, Routing may provide information about the optimality of each **next hop**, which can be used in the decision process.

Resource Allocator

The Resource Allocator takes the output provided by the Routing component and provides information to the PDU Forwarding function (for example PDU Forwarding table entries if this function is table-based). The RA can be distributed, collaborating with other RAs in members of the DIF (autonomic approach). The RA can also collect and communicate information to the DIF Management System (traditional approach). The traditional approach is suitable when resources in the members of the DIF are tightly constrained, while the autonomic approach is more suitable when fast reaction times are required. In practice, the norm will be somewhere in between.

There are basically three sets of information available to the RA to make its decisions:

- The characteristics of the flows requested by the applications using the DIF via "allocate flow requests".
- The properties of the N-1 flows the DIF is relaying on.
- Information from other members of the DIF on what they are observing (this latter category could be restricted to just nearest neighbors or some other subset - all two or three hop neighbors - or the all members of the DIF). This is carried out by means of RA-RA communications.

Generally speaking, 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 Resource Allocator, which includes aspects related to resource allocation policies**. These resource allocation aspects are within the RA’s scope, and may be determined by means of RA-RA communications.

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 and PDU forwarding table

This policy is invoked per PDU in order to obtain the N-1 port(s) through which the PDU has to be forwarded. It takes forwarding decisions on a PDU-by-PDU basis, so the delay overhead posed by its execution is a key requirement for a proper forwarding operation.

The PDU forwarding policy consults the PDU forwarding table, which stores the pre-computed forwarding information. In this way fast forwarding decisions are assured without involving additional computing cost. However, the forwarding policy may use additional dynamic information (e.g., queue residency times for the different (N-1) ports) along with the pre-computed forwarding table. Besides, the policy might make decisions in some other way than simply blindly using the pre-computed data (e.g., apportioning the traffic to different next-hops using a randomizing selection algorithm), but always complying with the tight time requirements.

In the simplest case, the PDU forwarding policy only needs to be passed the address and QoS cube from the PDU to make its forwarding decisions. However, in a multipath scenario the PDU forwarding policy must make the decision on what available path use to forward the traffic. In this case, other information in the EFCP PCI like CEP-ids might be useful to reduce the incidence of getting PDUs out-of-order at the destination.

2.2.5. A QoS-aware multipath solution for RINA focused on Datacenter Networking

In this section we describe the QoS-aware multipath routing policies for the DatNet use case. We first start by presenting the DIF structure, and then we proceed to specify the policies that apply for the following incremental routing strategies:

- Simple link-state routing, applying Dijkstra's algorithm to compute the shortest path.
- Multipath link-state routing, computing a set of different shortest paths and load-balancing the traffic among them.
- QoS-aware multipath link-state routing, enhancing the approach of the previous point by considering an intelligent QoS-aware multipath strategy to split the traffic among all the possible paths optimizing resource allocation while improving the network efficiency.

We follow this approach for the sake of the explanation's simplicity. We start by a simple link-state routing strategy and evolve it to achieve the QoS-aware multipath routing strategy for RINA, which is the one that will be finally implemented in the prototype.

The routing strategies in RINA are implemented by means of the following policies:

- Routing
- PDU forwarding table generator (RA)
- PDU forwarding policy (RMT) and PDU forwarding table (RMT)

DIF architecture

We will consider as the datacenter DIF architecture the configuration depicted by the following figure. The DC Fabric DIF spans among all the routers of the datacenter network, and the Tenant DIF connects VMs, servers and border routers with the exterior of the DC.

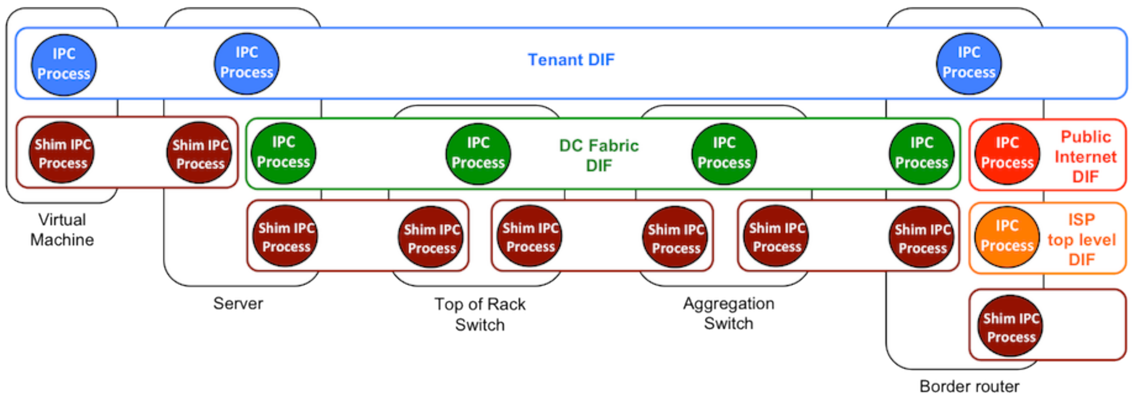


Figure 27. DIF architecture for Datacenter Networking

The routing along the datacenter network is performed within the DC Fabric DIF, which holds the connections among all the datacenter network nodes. The connectivity graph of the DC Fabric DIF is depicted in the following figure.

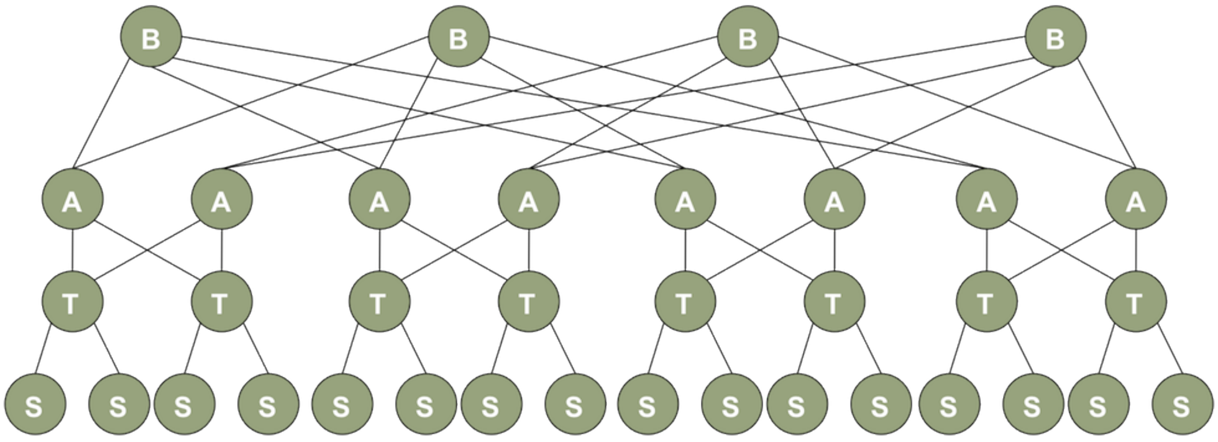


Figure 28. Connectivity graph of the DC Fabric DIF (S=Server, T=Top of Rack Switch, A=Aggregation Switch, B=Border Router)

Simple link-state routing

To start defining the policy design for the QoS-aware multipath routing strategy, let's first start by analyzing a non-multipath strategy such as a simple link-state routing strategy, which is based on the following process:

1. Each network node determines what other nodes it is connected to and through what ports. It does this using a "reachability protocol" which runs periodically and independently with each of its neighbors.
2. Each node periodically (and in case of connectivity changes) creates a "link-state advertisement" message containing:
    - The node which is sending the message.
    - The nodes to which the sender is directly connected.
    - A sequence number, which is increased every time the source node sends a new link-state advertisement.
3. The message is flooded throughout the entire network. Each node then stores the message with the more recent sequence number, so the last link-state advertisement quickly gets stored in every node in the network.
4. Each node calculates the network graph iterating over the set of link-state advertisements from each node of the network. For a link to

be considered as correctly reported the two ends must have sent the corresponding link-state advertisement. To that end, each node holds two data structures:

- A "network tree" containing the self node as root and the other nodes as branches and leaves following the "shortest-path" route from the node. The routing table is formed with the output port that identifies the branch that contains the destination node.
- A set of candidate nodes to be included in the tree, which is iterated to fill the tree accordingly.

5. The link-state message is recomputed and flooded again throughout the network whenever there is a change in the connectivity between the node and its neighbors (e.g link fail) detected by the reachability protocol.

The mapping of the functionality among the different policies for the link-state routing is the following:

- PDU forwarding table generator (RA)
    - Sending/receiving network information by means of CDAP messages (link-state advertisements), which in RINA would be flow-state advertisements. The state of the N-1 flows are stored in the so called FlowStateObjects, which are maintained by the FlowStateDatabase en each IPC process. Each FlowStateObject has an associated Age, whose maximum determines when the FlowStateObject is deprecated deleted. This FlowStateDatabase is the data exchanged between IPC processes by means of the CDAP messages when one or more FlowStateObjects have been modified but not yet propagated. It is sent to all the other IPC processes in the DIF. If the modification arrived from another IPC process, the FlowStateDatabase is not sent back to that IPC process.
    - Updating and maintaining the network graph based on the exchanged CDAP messages and internal events (i.e. N-1 Flow Allocated, N-1 Flow Deallocated and Neighbor Added) received from the Resource Allocator.
    - Computing and updating the PDU forwarding table. This is a periodic process that takes place if the FlowStateDatabase has been modified since the last PDU forwarding table update.

    - Killing FlowStateObjects. When a FlowStateObject has been deprecated, it is erased from the FlowStateDataBase. This "delayed removal" assures that the FlowStateObject is propagated before being erased.
    - Increasing the FlowStateObject's age in the FlowStateDatabase periodically.
- Routing
    - Implementation of the routing algorithm (e.g. Dijkstra algorithm)
    - Computing the PDU forwarding table from the network graph
- PDU forwarding policy (RMT) and PDU forwarding table (RMT)
    - Forwarding decisions on a PDU basis from the data stored in the PDU forwarding table, which in this case is a simple table containing mappings of <destination, QoS-id> tuples to outgoing ports.

Simple Multipath link-state routing

The next incremental step is to study a multipath strategy derived from the above link-state strategy. A simple case to consider is a multipath strategy which load-balances traffic among a set of pre-computed shortest paths. The load-balancing functionality, which is within the scope of the PDU forwarding policy, can be defined as a policy, leaving the door open for the implementation of different forwarding strategies among the set of shortest paths. In case the traffic is evenly distributed among them, it will be similar to the Equal Cost Multipath (ECMP) solution typically used in datacenter networks. However, different policies can be implemented in the PDU forwarding policy to spread the traffic among the paths with different distributions.

In our case, for the sake of simplicity, and focusing the feasibility of the multipath strategies in RINA, we will focus on a ECMP-like case. That is, distributing the traffic evenly among the set of shortest paths. ECMP does so distributing the TCP connections. In this case, this simple multipath routing solution for RINA will distribute PDUs belonging to different EFCP connections through the multiple N-1 flows.

In this case, the changes of the functionality of the different policies with respect to the previous case is the following:

- PDU forwarding table generator (RA)
  - idem.
- Routing
  - In this case, it is needed a routing algorithm that retrieves the set of shortest paths. The most direct approach is to consider a "shortest path tree" computed using the Dijkstra algorithm over the network graph.
  - As for the computing of the PDU forwarding table from the network graph a different approach is needed, since in this case not only a single path is returned, but the set of all the shortest paths.
- PDU forwarding policy (RMT) and PDU forwarding table (RMT)
  - In this case, the PDU forwarding policy distributes the traffic evenly among the different paths. Therefore, the PDU forwarding policy must implement the necessary mechanisms for that purpose, retrieving the available ports from the PDU forwarding table and distributing the PDUs among them (e.g. in a Round-Robin fashion for the ECMP case).
  - In this case the PDU forwarding table needs to account for the extra information needed to perform multipath. To that end multiple entries for the same <destination, qos-id> pairs will account for the different possible next hops.

QoS-aware multipath link-state routing

The previous studied routing strategies (simple link-state and simple multipath link-state) only consider the link state (up or down) and a cost metric (number of hops) to determine the shortest paths and to form the DPU forwarding table accordingly. Here we propose a QoS-aware multipath routing solution that considers also the QoS requirements of the flows to take the forwarding decisions, i.e., the QoS requirements of the N flows are considered to forward the traffic through the N-1 flows.

Two different possibilities are considered depending on whether dynamic real-time link-state (or flow-state) information is available (e.g. link utilization/congestion level, etc.).

Static QoS-aware multipath routing

In the static QoS-aware multipath case, no dynamic link-state information is used and the only information available per-link is:

- Whether the link is up or down
- Associated cost metric
- Link bandwidth

In this case, the only information to make "QoS-aware forwarding decisions" is to consider the known QoS requirements of the N-flows to generate the PDU forwarding table. As a first approach, we will consider the QoS requirements on delay and jitter, and the average bandwidth of the flow as input to the routing algorithm.

Many possibilities are present with regards the routing algorithms in this case. There is a wide range of possibilities when splitting traffic among different paths considering the traffic characteristics. As a first step for static QoS-aware multipath routing algorithms, we propose a sort of "QoS-driven load-balancing algorithm", in which the flows are spread among the different shortest paths aiming at two objectives:

1. Not overloading the paths. That is, if we have a set of flows with high average bandwidth and low average bandwidth, the goal is to avoid sending over the same paths the bigger flows, therefore combining over the same paths big and small flows so that the paths do not get congested. In the same way, if no feasible way do exist to spread the flows among paths so that none of them gets congested, the goal is to minimize the caused overload, constraining it to the minimum number of paths.
2. Sending the data belonging to the same flow over the same path. Thus avoiding reordering delays caused by multipath.

Note that the approach proposed here should not be the optimal one, since other algorithms may (and most likely do) have better performance. However, we leave this aspect for the second part of the project, and for this deliverable we focus on the realization of a feasible static QoS-aware multipath algorithm.

In this case, the changes of the functionality of the different policies with respect to the previous case is the following:

- PDU forwarding table generator (RA)
  - Idem.
- Routing
  - Idem as in the simple multipath case.
- PDU forwarding policy (RMT) and PDU forwarding table (RMT)
  - In this case, the PDU forwarding policy distributes the traffic among the different paths according to the flows' QoS requirements. To that end, the PDU forwarding policy in each IPC process must store a record of the following:
    - Connection to what the PDUs that have reached the IPCP belong to. This is identified by the connection endpoints ids (cep-src-id, cep-dest-id) and the qos-id. This is done to forward the PDUs belonging to the same flows through the same port. Each time an unidentified PDU arrives, the connection identifiers are stored and a port is chosen for that PDU. For the next PDUs belonging to the same connection, they are forwarded through the same port. A time span is associated with each of the connections, and updated periodically. When no PDUs of a certain connection have been forwarded over a certain time span, the connection register is deleted.
    - Average bandwidth allocated to each port. Each time a port is chosen to forward a PDU, the PDU forwarding policy stores the accumulated average bandwidth of the connections allocated to that port.
  - The QoS-aware multipath algorithm implemented by the PDU forwarding function consists on the following main steps:
    i. When a PDU of a new connection arrives at the IPC process, the PDU forwarding policy checks the set of paths that lead to the destination and retrieves the set of ports that can be used to forward the PDU.
    ii. The port with less average bandwidth allocated is chosen, the connection identifiers are stored, and the average bandwidth allocated for that port is incremented.

iii. For the next PDUs of that connection, the PDU forwarding policy checks that it has the connection identifiers already stored, and forwards the PDUs through the already allocated port.

iv. When a connection reaches the maximum time without PDUs being forwarded through that IPC process, the connection identifiers are deleted and the bandwidth allocated to the corresponding port is reduced.

- The PDU forwarding table does not change with respect the previous simple multipath case.

Dynamic QoS-aware multipath routing

In the dynamic QoS-aware multipath case, dynamic link-state information is available in addition to the previous static per-link information. In this case the available information per link is:

- Whether the link is up or down
- Associated cost metric
- Link bandwidth
- Dynamic monitored information
  - Instant traffic load/link utilization (where "instant" refers to the last monitored value)
  - Traffic statistics collected at network level.

In this case, apart from considering the QoS requirements of the N-flows, the dynamic monitored information can be also used to generate the PDU forwarding table.

Many possibilities are also present in this case with regards the routing algorithms. The previously proposed "QoS-driven load-balancing algorithm" for the static QoS-aware multipath routing case, can be extended to consider the traffic stats per link and spread the flows among the different shortest paths aiming at the same two objectives of the above point:

1. Not overloading the paths. Apart from combining over the same paths big and small flows, the Dynamic QoS-aware multipath routing

approach aims to consider the congested links to avoid forwarding traffic through them, choosing alternative (and less congested) paths when this situation occurs.

2. Sending the data belonging to the same flow over the same path. Thus avoiding reordering delays caused by multipath.

In this case, the changes of the functionality of the different policies with respect to the previous case is the following:

- PDU forwarding table generator (RA)
    - The Resource Allocator, by means of the PDU forwarding table generator may use traffic statistics and generate the PDU forwarding table accordingly. For example, if a highly congested link is detected, the PDU forwarding table entries forwarding traffic through that link may be deleted, thus eliminating the possibility of routing traffic through it. This decisions are to be made on a coarse time granularity, i.e., when a link is congested for large periods of time.
- Routing
    - Idem as in the static QoS-aware multipath case.
- PDU forwarding policy (RMT) and PDU forwarding table (RMT)
    - As in the static case, the PDU forwarding policy distributes the traffic among the different paths according to the flows' QoS requirements. To that end, the PDU forwarding policy in each IPC process also stores a record of the connection to what the PDUs that have reached the IPCP belong to, identified by the connection endpoints ids (cep-src-id, cep-dest-id) and the qos-id. Note that in thics case the average bandwidth allocated to each port is not relevant and it’s not stored.
    - In this case, the dynamic QoS-aware multipath algorithm implemented by the PDU forwarding function consists on the following main steps:
        i. When a PDU of a new connection arrives at the IPC process, the PDU forwarding policy checks the set of paths that lead to the destination and retrieves the set of ports that can be used to forward the PDU.

        ii. The monitored traffic statistics are consulted per port, and the one with less traffic load is chosen. Then, the connection identifiers are stored to forward the forthcoming flow PDUs through the same port.

iii. For the next PDUs of that connection, the PDU forwarding policy checks that it has the connection identifiers already stored, and forwards the PDUs through the already allocated port.

iv. When a connection reaches the maximum time without PDUs being forwarded through that IPC process, the connection identifiers are deleted.

- The PDU forwarding table does not change either with respect the previous simple multipath case.

3. Topological addressing to bound routing table sizes

3.1. Introduction

This chapter focuses on the specification of the addressing and routing architecture in the scope of PRISTINE project. Basically, it proposes generic architectures in line on one hand with the requirements of the PRISTINE use cases (from WP2, presented in deliverables [D21] and [D22]) and compliant on the other hand with the RINA architecture. In order to support scalability, which is the main constraint in all PRISTINE use cases, the proposed architectures rely on the concept of “divide and conquer” provided by the recursive nature of RINA. As a first evaluation, this document presents some initial simulation results that have been conducted using RINASim simulator.

3.2. Routing and Addressing in PRISTINE Use Cases

3.2.1. Distributed Clouds Use Case

Characteristics and requirements of the use case

VIFIB is a decentralized cloud system, also known as resilient computing. It consists of computers that are located in people’s home, in offices, etc. VIFIB uses an overlay called re6st, which creates a mesh network of OpenVPN tunnels on top of several IPv6 providers and uses the Babel protocol for choosing the best routes between nodes. PRISTINE will provide an alternative to the re6st overlay, by using RINA on top of IPv6. The re6net overlay organizes nodes in a flat random graph, using its own algorithm. The goal of this algorithm is to construct a robust network structure with a small diameter, in order to minimize the latency between nodes. Since the routing tables of the overlay are under the control of VIFIB -and not the ISPs-, the overlay can recover faster from a link failure than BGP or other algorithms used by Internet providers. The system should provide higher resiliency in order to avoid losing connectivity and higher privacy.

PRISTINE will develop an alternative strategy to re6st. With the RINA architecture, the cloud participants are seen as application processes which use different DIFs to communicate together. The system must be able to provide high security level and ensure certain level of QoS (delay and loss). Moreover, the routing approaches that will be applied should cope with the scalability requirement of the distributed cloud system. The main characteristics of this use case that should be taken into account could be summarized into:

- All ViFIB nodes can be routing and application nodes at the same time. VIFIB nodes are distributed around the world and do not have a complete knowledge about the network connectivity. Moreover, VIFIB nodes shall build an overlay to help each other to find the best possible route to interconnect two services on two edges. Accordingly, a VIFIB node could act as an edge sometimes and in other as a router to a service on other VIFIB node. Solutions that rely on recursion in order to reduce the addressing space by hiding relay nodes do not provide any benefit here.
- All ViFIB nodes are located at customers' machines and with limited bandwidth resources. Solutions that rely heavily on recursion in order to make the addressing space scalable should not be used, as aggregated flows require large bandwidth in lower layers.
- Reliability is a key concern of the cloud. Any solution considered must take into account the possibility of losing multiple ViFIB nodes at the same time.
- After making it reliable, low delay is desired. Proposed solutions should provide a way to minimize latency between pairs of ViFIB nodes.

In the next sections, we will consider the requirements and the restrictions of the distributed clouds use case in order to design efficient addressing and routing policies.

Applying RINA to the use case

System Management Architecture

The Distributed cloud system is maintained by a management entity which provides the initial point of contact for nodes joining the overlay system. Moreover, nodes should keep updated with this entity for maintenance reason. Accordingly, we assume that all the VIFIB infrastructure is a single management domain the Network Management Distributed Management System (NM-DMS). We also assume a scenario in which there is logically centralized Manager process configuring and monitoring the VIFIB nodes via management agents deployed at each node. The Manager process can communicate with the Management Agents via a separate DIF, dedicated to the NM-DMS as depicted in Figure 29.

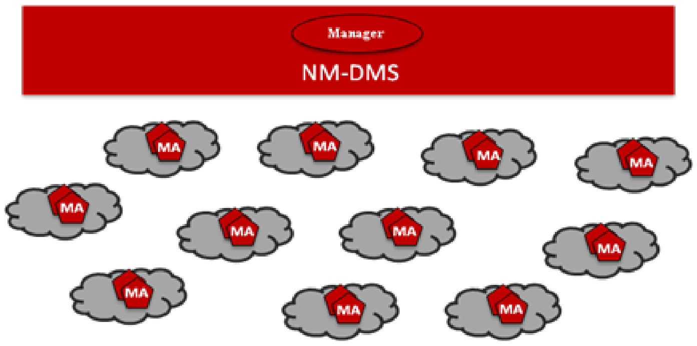


Figure 29. DAF/DIF configuration of Distributed clouds system

Services provided by the distributed cloud system are deployed using App-DAFs. App-DAF is a collection of Distributed Application Processes (DAPs) that will be sharing information using specialized overlay App-DIFs that are tailored to the needs of the App-DAFs as illustrated in Figure 30. Below, there will be DIFs managing the routing between the DAPs. In the next section, we will address the global architecture of DIFs ensuring the exchange of data between the DAPs. Routing policies and addressing are specified.

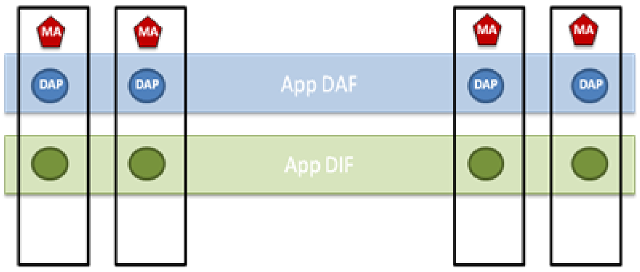


Figure 30. DAF/DIF configuration of Distributed clouds system

Addressing and Routing policies

Routing in distributed clouds system could be tackled from different perspectives and with different approaches. We propose in the scope of PRISTINE two distinct concepts with the same aim to bound the routing table size:

- A generic solution, called SFR (Scalable Forwarding in RINA), which is meant to cope with large-scale distributed clouds scenarios (+10.000 ViFIB nodes) building a hierarchical DIF architecture.
- A second solution based on addresses aggregation which is adapted more for medium scenarios (~1.000/10.000 ViFIB nodes) and where we suppose that VIFIB nodes could be managed within a single DIF.

SFR: Scalable Forwarding in RINA

When dealing with distributed clouds system, we need to consider that in such large-scale scenario increasing bandwidth usage could affect the performance of the Cloud, as resources are very limited. Accordingly, instead of focusing on having a large Cloud where any pair of nodes is connected, we propose to apply the "divide and conquer" concept and have a hierarchy of smaller clouds providing connectivity between the pairs of the system in an efficient way. The main idea of SFR is to divide the clouds into groups or regions. These groups are created and managed by the authorities based on a specific criterion that could be the group size, the country or the ISP membership.

Furthermore, connectivity between the groups is ensured by inter-connecting a set of VIFIB nodes of each group. This set of VIFIB nodes, namely "groups leaders", are elected to act as relays between the groups and to form specifically the inter-groups. At the same time they preserve their membership to their original groups. As depicted in Figure 31, in order to cope with the scalability, this "logical" organization could be repeated recursively adding other levels that will be forming a logical hierarchy. To avoid link failure, several VIFIB nodes could be elected as group leaders. The way they are chosen needs further investigation.

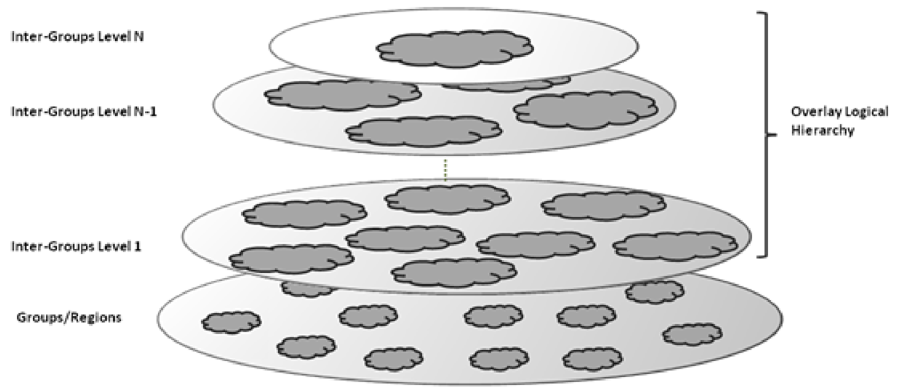


Figure 31. Hierarchical routing architecture for distributed clouds use case

Let's take an example of three levels of Inter Groups Hierarchy as illustrated in Figure 32. Group S is the group where the originating VIFIB node A belongs. Group D is the group where is the destination VIFIB node (H).

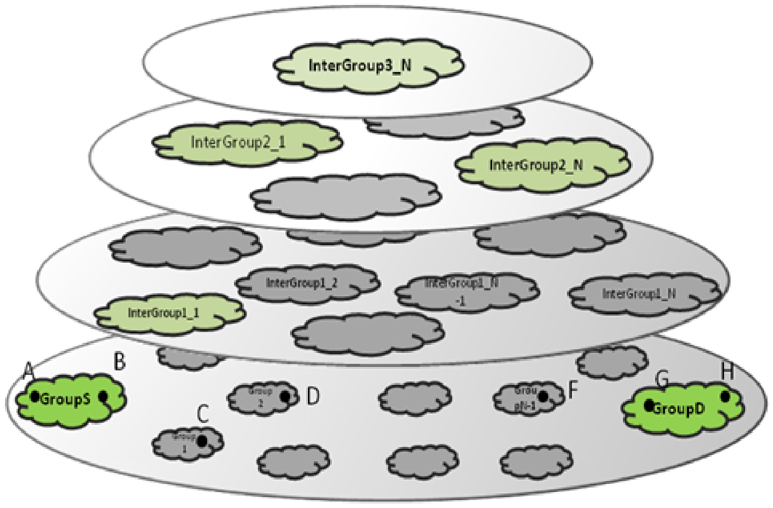


Figure 32. Example of SFR hierarchy

To represent this scenario in RINA logic, we draw Figure 33. We assume that for each region/group a DIF is created to manage connectivity inside the group. Consequently, Each VIFIB node has at least one IPCP in the groups of the overlay (the lower level of the hierarchy). Some of the overall VIFIB nodes that we called "group leaders" will have also IPC Processes in the inter-groups on the upper logical levels apart from the IPCPs belonging to the Group DIF.

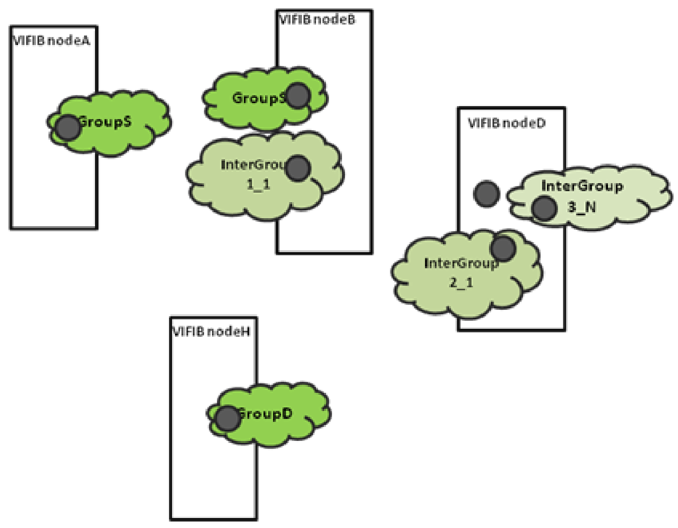


Figure 33. Example of number of IPCPs in each VIFIB node

Suppose that VIFIB node A in Group S is the source node and node H in Group D is the destination. Node A has one IPCP connected to the groupS DIF which connect to node B that acts here as the group leader. Accordingly, VIFIB node B has one IPCP within Group S and one additional IPCP within InterGroup1_1 that connect it to node C in the scope of the Inter Group 1_1. Node C has 3 IPCPs: One within Intergroup 1_1, one within InterGroup2_1 and at the same time one within a Group1 in the lower group DIFs. Node D has 4 IPCPs: One within Intergroup 2_1 at level 2 allowing connection with the node C, one within InterGroup3_N at level 3 to connect to node F. Moreover, it has an IPCP at level 2 in InterGroup1_2 and another one within Group2 in the lower level group DIFs. On the other side, VIFIB Node G has 3 IPCPs and in particular one in GroupD where the destination VIFIB node H belongs. Accordingly, node G will use the Group D DIF to reach directly the destination instead of following the hierarchy of DIFs level by level.

In Figure 34 we illustrate the DIF architecture of the overlay cloud in the considered example.Each group is mapped to a DIF which is created accordingly to manage connectivity inside this region. Moreover, as we mentioned in Section 3.1.2.1, each App-DAF has a “tenant App-DIF” as stated in the figure. This DIF is destined to connect DAP1 and DAP2 in order to support their communication process. The tenant Cloud DIF is designed to adapt to the dynamic network connectivity. Especially, for the distributed cloud use case, VIFIB node could act as Border routers and at the same time as customer’s app. The tenant Cloud DIF ensures flexibility and maintains a global view of the network as a full mesh to manage dynamically the possible suppression/appearance of the lower DIFs structure which could be very frequent in the distributed clouds scenario. To summarize, we have basically 4 types of DIFs:

- **Group DIFs**. Small DIFs (~100 nodes) that provide high connectivity and low latency between small groups of nodes.
- **Inter-group DIFs**. Small DIFs (~100 nodes) that provide connectivity to group leaders of some Group DIFs.
- **Tenant Cloud DIFs**. Medium-sized DIFs (~1.000/10.000 nodes), that provide connectivity between VIFIB nodes of the different group DIFs whose group leaders share the same Inter-group DIF. These DIFs could be created dynamically on demand.

- **Tenant App DIFs**. DIFs that provide the direct connectivity between hosts. Mainly it is used by a customer of the Distributed Cloud. These DIFs are directly supported over a tenant Cloud DIF.

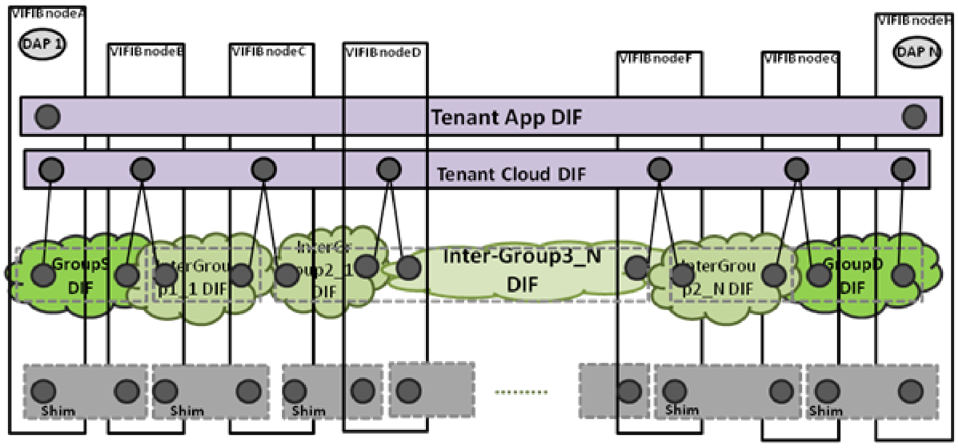


Figure 34. SFR DIF Architecture. Hierarchical Routing in RINA

Managing dynamically the creation of "Tenant Cloud DIF"

In the case of the distributed clouds use case, some DIFs could be pre-configured to support the connectivity between customers that are frequently communicating. However, DIFs could be created when it is needed which means that when resources are requested to be allocated between source and destination, the DIF allocator will be in charge to build (if not existing already) the required Tenant cloud-DIFs to ensure the connectivity. If a customers' VIFIB node do not find a common DIF where it can see the destination, it should query peer DIFs which may have a DIF with the destination Application process. The path followed by the search request will be the sequence of the DIFs to use given by the DIF Allocator forwarding table.

- If two VIFIB Nodes in the same region want to communicate, they don't need to be a part of more than one DIF. As an example, see Figure 35, DAP1 in node A and DAP 2 in node B can communicate directly using "GroupS DIF". So, only an App-DIF is created to support the communication.
- In Figure 36, the VIFIB nodes A and C are not in the same region. The creation of a tenant cloud DIF is crucial. Three IPCPs are created on node A, node B (the border router) and node C.

- In the third example in Figure 37, the same tenant cloud DIF will be extended to cover the nodes leading to DAPN in node H: node F, node G and node H will be added to the DIF.

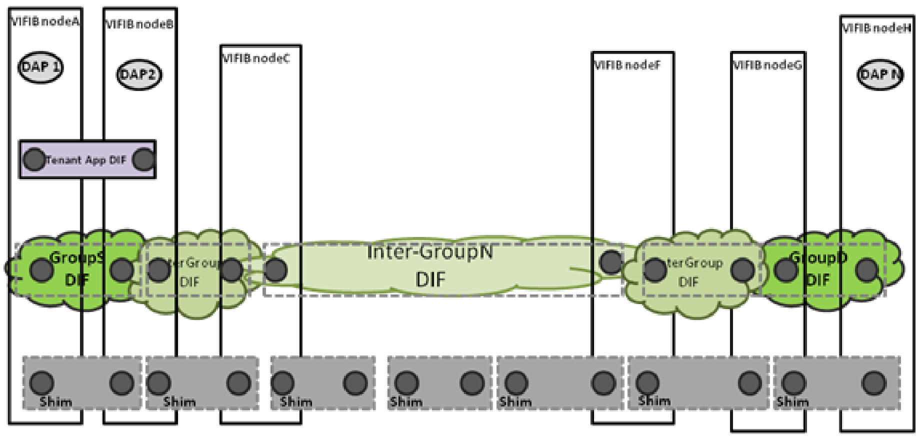


Figure 35. VIFIB nodes in the same region

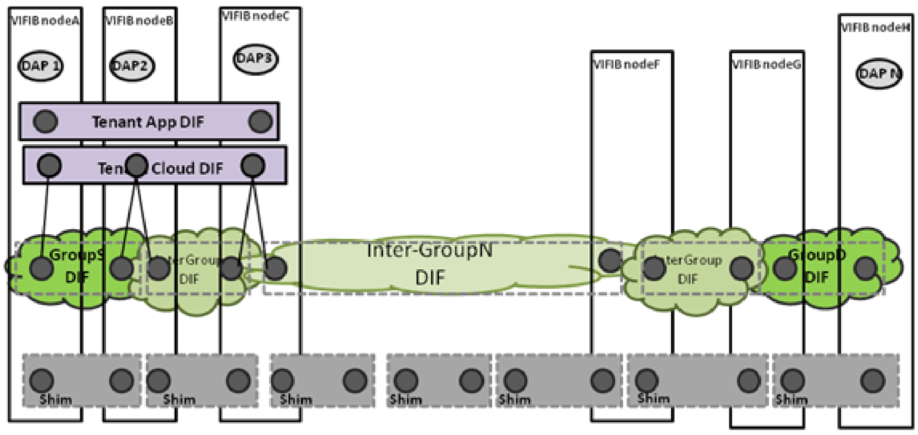


Figure 36. VIFIB nodes in different close regions

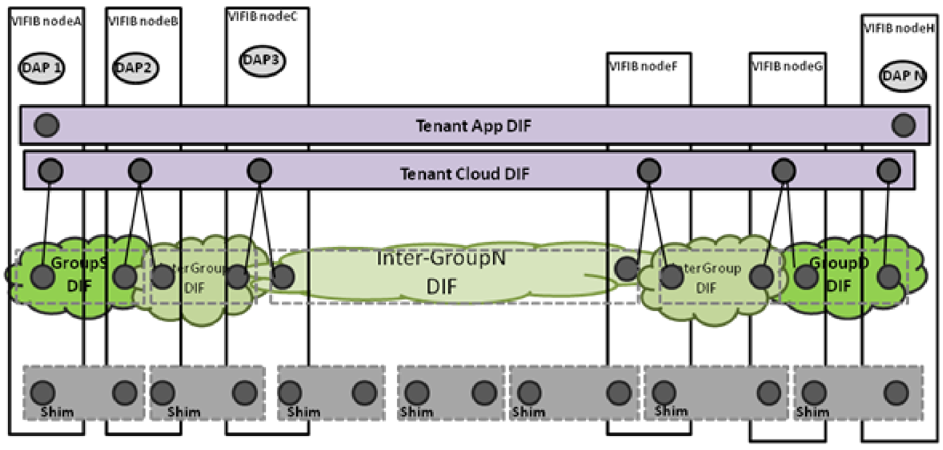


Figure 37. VIFIB nodes in different far-away regions

Routing Policies

The routing algorithm to be run in the DIFs will depend basically on the built DIF hierarchy. In the Groups DIFs at the lowest level of the hierarchy, VIFIB nodes have to store routes leading to the border routers. Consequently, traditional routing e.g. link state or distance vector could be used. Then, group leaders can determine the next hop based on topological addresses. Traditional routing might be used also inside the inter-groups if needed. In the next section, we will address the address assignment issue.

Topological Addressing

Figure 38 illustrates an example of the topological addressing that could be deployed for the distributed cloud system. Basically, each DIF has its own address space that is independent of the adjacent DIF. However, the upper branches of the hierarchy may have a topological relation with lower layers which could simplify routing calculations. Addresses belonging to the same authorities or located in the same geographic place would be similar. Moreover, for each layer in the hierarchy more granularities have to be provided.

In the example in Figure 38, in the lowest level of the hierarchy the address is built from country prefix, ISP prefix and number of nodes. At upper level (inter-groups), the country code is used as a prefix for the address (FR, TN). At the very upper level the address space cover the whole globe and there we can find VIFIB nodes belonging to different countries where addresses start with FR, TN and so on. Following the whole address, the needed path could be found. Accordingly, the topological address defines the concept of "nearness". This provides location dependence without route dependence.

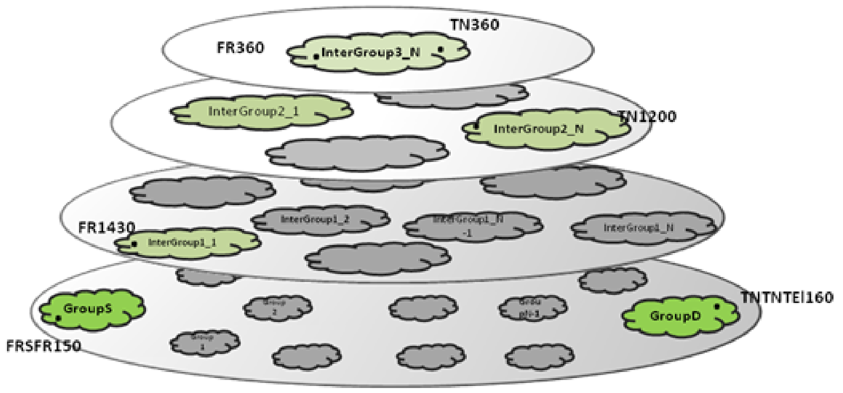


Figure 38. Topological Addressing Assignment Example

Address Aggregation Solution

In small network scenarios (~100 ViFIB nodes), quite simple solutions can already match. Following the current Nexedi solution, all nodes could be deployed in the same overlay DIF and a high number of connections can be used in conjunction with a simple distance vector or link state routing policy with latency as part of the metric, without adding constraints to the network.

However, when dealing with medium size Clouds (~1000/10000 nodes), the network size goes up, it is impossible to rely on a large number of connections between nodes in order maintain the diameter/latency in the network low, as that would not only increment the number of flows that each node has to manage, but also the number of updates that each node would receive.

Moreover, although routing tables would be still manageable with 1k-10k nodes, reducing them at this point is already desirable, even at expenses of light sub-optimal routing. In this context, we propose two different methods that aggregate addresses given a common prefix. Specifically, we limit the configurations to only one level of aggregation in order to reduce the bandwidth of possible resulting aggregated flows.

Aggregation over the same DIF

Following the idea of the small Cloud, small groups of ViFIB nodes of the same DIF (called sub-DIFs) are created and managed by a centralized entity, ensuring that these sub-DIFs maintain a small diameter and low latency see Figure 39. A prefix is given to each sub-DIF and addresses in the form of <prefix.identifier> are given to each node.

Connectivity decisions between the different sub-DIFs are done abstracting these sub-DIFs as a full node, and any connection between nodes on two distinct sub-DIFs as a connection between sub-DIFs. Since there are no border routers and each sub-DIF has a similar number of VIFIB nodes as the number of sub-DIFs, if shared, any node only needs to have few connections to other prefixes in order to ensure “direct” connectivity between any pair of sub-DIFs.

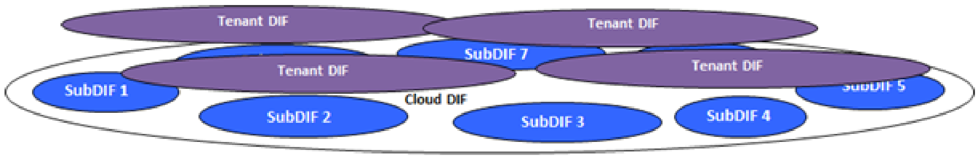


Figure 39. Global Cloud DIF divided into sub-DIFs, connected within the same DIF

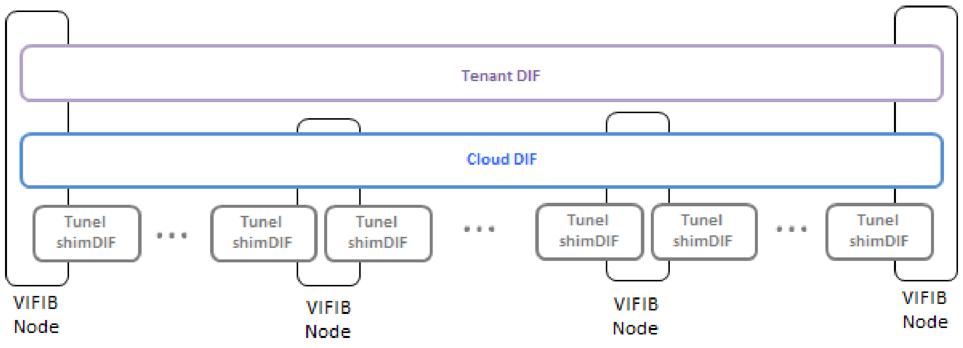


Figure 40. RINA stack. Tenant DIF over the Cloud DIF

Forwarding in the Cloud is done in two steps:

- If current node is in distinct sub-DIF than the destination node (dst), forward to the sub-DIF where dst belongs.
- If current node is in the same sub-DIF as dst, forward to dst.

Given that we only store information about the nodes in our sub-DIFs and how to reach other sub-DIFs, routing tables become really small (~100/250 entries). In addition, given the high connectivity in sub-DIFs and between them, we can ensure small paths and high reliability in the network. The two defined routing domains (intra sub-DIFs and inter sub-DIFs) does not need to use the same routing algorithm to compute forwarding entries. Instead of that, given the distinct properties of each domain, we propose to maintain inside sub-DIFs a link-state routing (in the same way as for small Clouds), while using a distance-vector algorithm between sub-DIFs.

Aggregation via recursion

Following the previous approach, small groups of ViFIB nodes are created and managed by a centralized entity and the same address space is used (addresses in the form of <prefix.Id>). In contrast with the previous solution, in this case, at each sub-DIF, some nodes are selected as border routers, and only border routers are used to interconnect the distinct sub-DIFs. In order to improve reliability, isolation and allow changes in the connectivity between border routers, a backbone DIF is created containing only border nodes. This backbone DIF is used by the Cloud DIF in order to communicate distinct sub-DIFs, as shown in Figure 41.

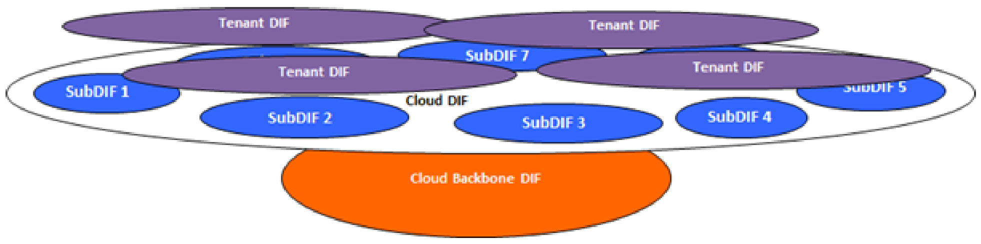


Figure 41. Global Cloud DIF divided into sub-DIFs, connected via a Cloud Backbone DIF

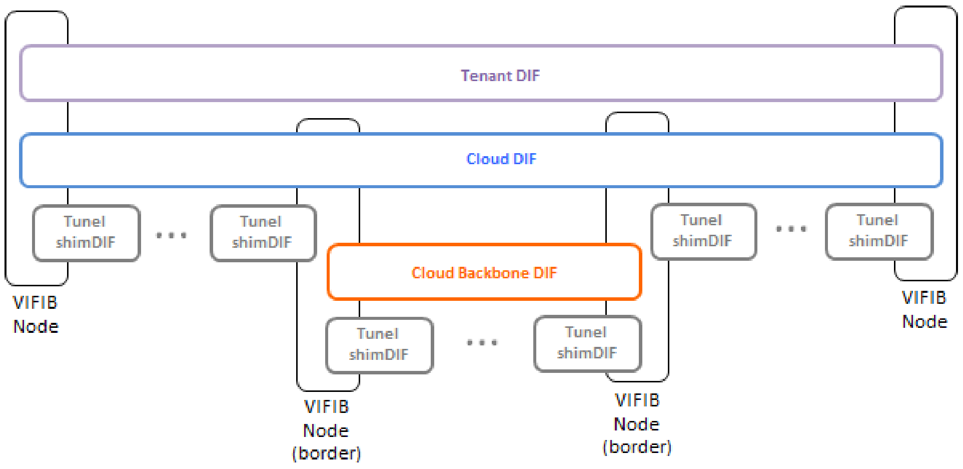


Figure 42. RINA stack. A tenant DIF over the Cloud DIF. Cloud Backbone used to communicate sub-DIFs of the Cloud DIF

Forwarding in the Cloud is done in 3 steps:

- If current node is in a distinct sub-DIF than dst, and current node is not a border, forward, within the Cloud, towards any border node of the sub-DIF.
- If current node is in a distinct sub-DIF than dst, but is a border, forward, within the backbone, towards any border node of the dst sub-DIF
- If current node is in the same sub-DIF as dst, forward towards dst via the Cloud DIF.

Given that, non-border nodes only need to store routing entries towards nodes in the same sub-DIF. Border nodes in this case, in addition to the entries to the nodes in the same sub-DIF, they have to store entries to all the other borders in the backbone DIF and maintain an operative flow to at least one border of all the other sub-DIFs, in order to have inter-sub-DIF connectivity over the Cloud.

Performance Evaluation

In this section, we evaluate the proposed approaches for addressing and routing in PRISTINE using RINASim. RINASim is the simulator being developed in the scope of PRISTINE project. It is intended to enable the study of the properties of RINA by means of intrinsic mechanisms or policies and also to perform simulation experiments with RINA applications. In this section, we give some details about the routing policies that have been implemented within RINASim including distance vector and link state approaches. Then, we present the scenarios that have been designed for the distributed clouds use case. Last, we provide the simulation results.

Policies for routing & forwarding : Design and implementation details

In the current implementation, RINASim routing and forwarding functions are divided into 3 distinct policies:

- **Forwarding Policies**. Performs fast forwarding decisions based on PDU headers. Main function is, "lookup", returning the list of output ports to forward the PDU.
- **Routing Policies**. Performs slow routing computations based on partial information about links, neighbours, etc. (depending on the policy in use).
- **PDU Forwarding Generator Policies (PDUFG)**. In charge of populate the Forwarding policy and update the knowledge of the Routing policy. This policy is in charge of the managing the distinct routing domains, when dealing with sub-DIFs, and configure the Routing policy according to that.

The policies that have been implemented already in RINASim are listed below:

- **Forwarding Policies**:
    - *MiniTable*: Simple forwarding table for flat addresses.
    - *SimpleTable* and *QoSTable*: Simple forwarding table for flat addresses + QoS Cube.
    - *DomainTable*: Forwarding table based on domains defined by prefixes. Addresses are segmented using the dot symbol as separator,

and searches only try to match the next segment after the domain prefix. The order in which domains are defined is important, as lookup search in the PDU domain in the order in which those have been defined.

- **Routing Policies**:
    - *DummyRouting*: Default Routing policy for static scenarios that does nothing.
    - *SimpleLS* and *SimpleDV* (SimpleRouting family): Routing policies for Link State and Distance Vector routing over a flat addressing scheme. IPCP address is used as node address and flow info shared by routing algorithms includes dst address, QoS and metric.
    - *DomainRouting*: Routing policy for use with distinct domains (sub-DIFs, routes/metrics depending QoS, etc.). When defining a domain, the domain name, node address within the domain and routing algorithm to use (LS or DV) is given. Flow info shared by routing algorithms includes domain name, dst address and metric.
- **PDUFG Policies**:
    - *StaticGenerator*: Policy for static networks where routing info is read from config files.
    - *SimpleGenerator*: Policy for routing in flat addressing schemes.
    - *SingleDomainGenerator*: Policy for routing in flat addressing schemes.
    - *QoSDomainGenerator*: Policy for routing in flat addressing + QoS schemes.
    - *BiDomainGenerator*: Policy for routing in addressing schemes of inter/ intra sub-DIFs domains.

Scenarios in RINASim

In order to test the different policies in the scope of distributed clouds use case, two distinct scenarios have been created:

- **Distributed Clouds use case: small example**. Figure 43 depicts a small network composed of 4 regions/groups, each region with 3 VIFIB nodes including the group leaders (VIFIBNGL). The DIF architecture is organized as follows:
    - A Group DIF for each region.

  - An Inter-group DIF interconnecting region 1/region 2 and region 3/region 4.
  - An intergroup DIF interconnecting region 2/region 3.
  - A cloud tenant DIF contains all the nodes that are communicating.

- **Distributed Clouds use case: medium example**. Figure 44 represents a medium size network of ~120 nodes, divided into 4 regions, within each region 30 VIFIB nodes including the group leader. All the nodes inside the regions are interconnected randomly and connected to the group leader. All the group leaders are interconnected among each others. The DIF architecture is organized as follows:
  - A Group DIF is constructed to regroup all the VIFIB nodes inside each region.
  - Three inter-group DIF are designed to interconnect region 1/2 , region2/3 and region 3/4.
  - A cloud tenant DIF contains all the nodes that are communicating.

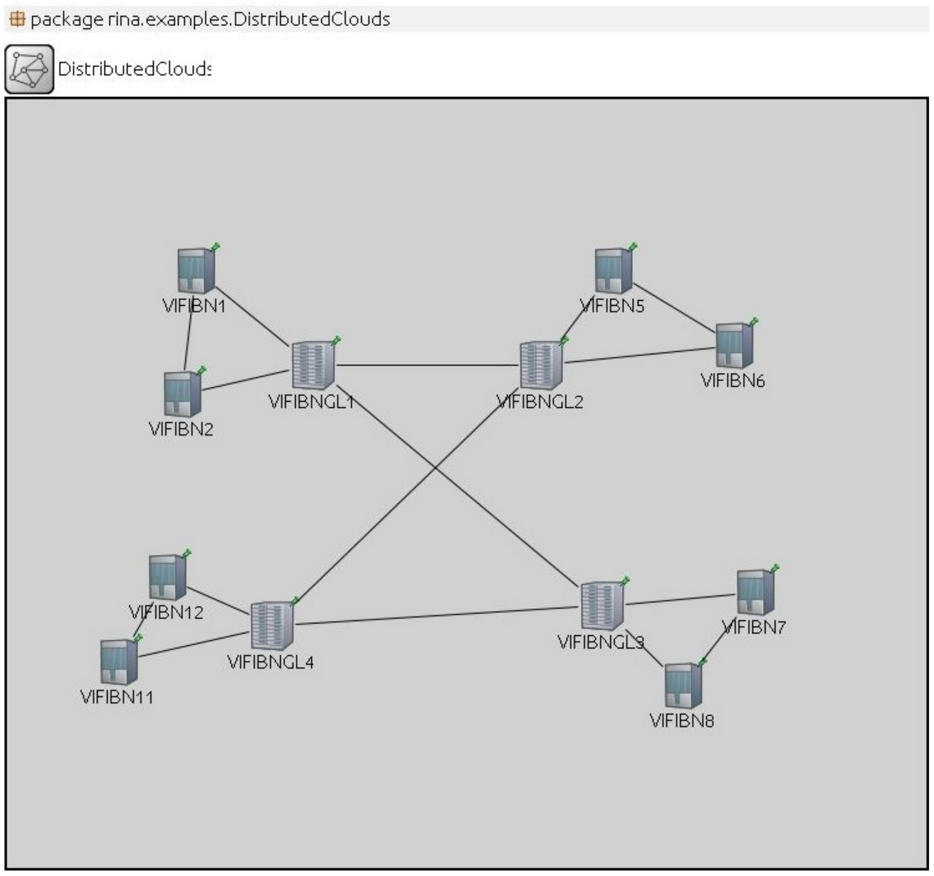


Figure 43. Distributed Clouds use case small example on RINASim

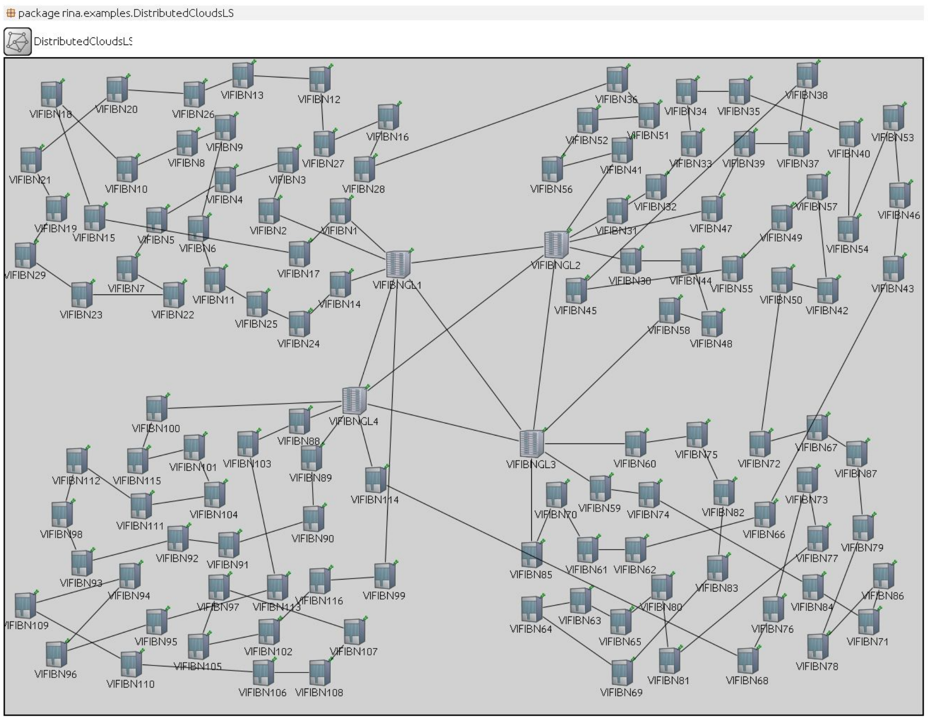
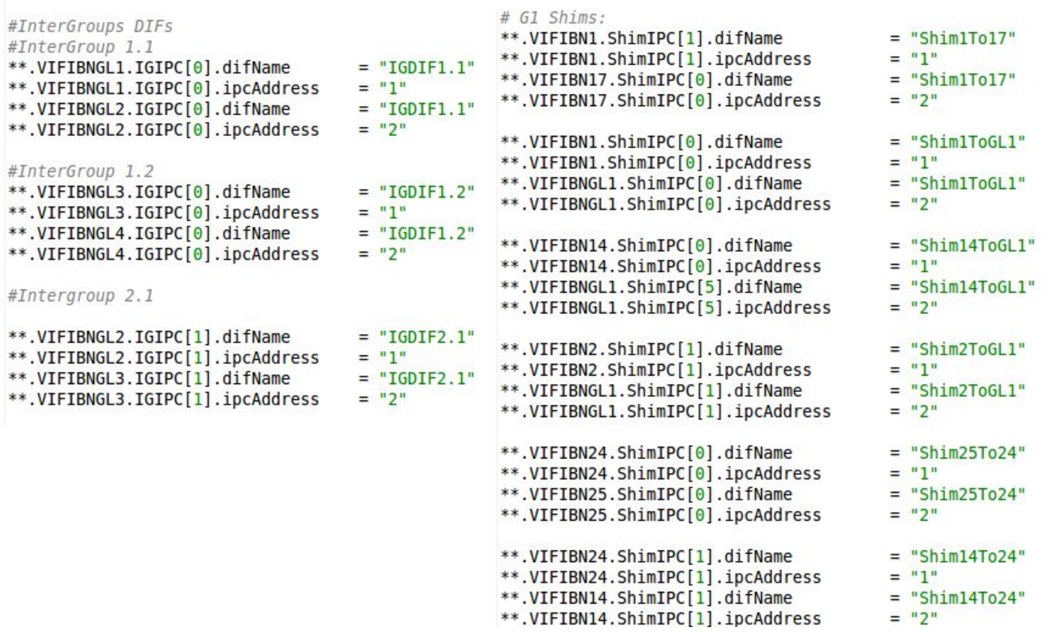


Figure 44. Distributed Clouds use case medium example on RINA sim

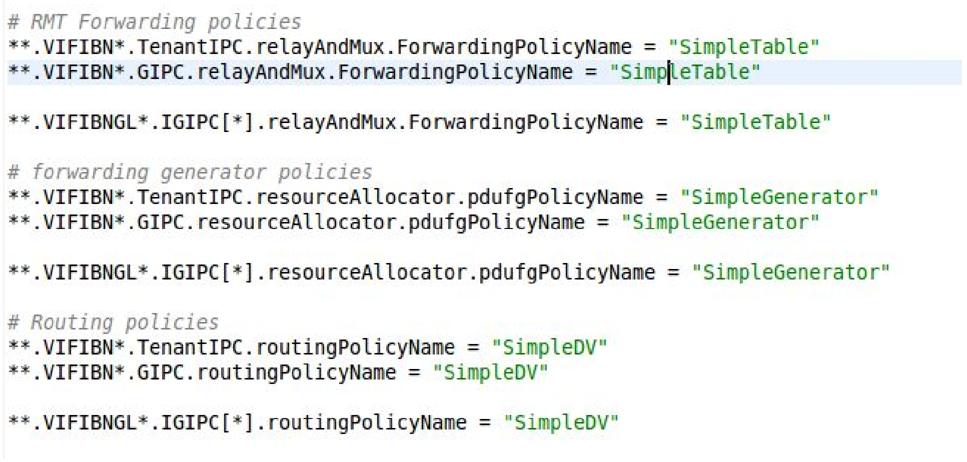
In both scenarios, policies behave as desired, resulting in a complete connectivity between nodes in the net. Distance vector routing policy has been used for routing inside the groups. Figure 45 and Figure 46 give more details about the configuration set using RINASim.



```
#InterGroups DIFs
#InterGroup 1.1
**.VIFIBNGL1.IGIPC[0].difName           = "IGDIF1.1"
**.VIFIBNGL1.IGIPC[0].ipcAddress        = "1"
**.VIFIBNGL2.IGIPC[0].difName           = "IGDIF1.1"
**.VIFIBNGL2.IGIPC[0].ipcAddress        = "2"

#InterGroup 1.2
**.VIFIBNGL3.IGIPC[0].difName           = "IGDIF1.2"
**.VIFIBNGL3.IGIPC[0].ipcAddress        = "1"
**.VIFIBNGL4.IGIPC[0].difName           = "IGDIF1.2"
**.VIFIBNGL4.IGIPC[0].ipcAddress        = "2"

#Intergroup 2.1

**.VIFIBNGL2.IGIPC[1].difName           = "IGDIF2.1"
**.VIFIBNGL2.IGIPC[1].ipcAddress        = "1"
**.VIFIBNGL3.IGIPC[1].difName           = "IGDIF2.1"
**.VIFIBNGL3.IGIPC[1].ipcAddress        = "2"
```

```
# G1 Shims:
**.VIFIBN1.ShimIPC[1].difName           = "Shim1To17"
**.VIFIBN1.ShimIPC[1].ipcAddress        = "1"
**.VIFIBN17.ShimIPC[0].difName          = "Shim1To17"
**.VIFIBN17.ShimIPC[0].ipcAddress       = "2"

**.VIFIBN1.ShimIPC[0].difName           = "Shim1ToGL1"
**.VIFIBN1.ShimIPC[0].ipcAddress        = "1"
**.VIFIBNGL1.ShimIPC[0].difName         = "Shim1ToGL1"
**.VIFIBNGL1.ShimIPC[0].ipcAddress      = "2"

**.VIFIBN14.ShimIPC[0].difName          = "Shim14ToGL1"
**.VIFIBN14.ShimIPC[0].ipcAddress       = "1"
**.VIFIBNGL1.ShimIPC[5].difName         = "Shim14ToGL1"
**.VIFIBNGL1.ShimIPC[5].ipcAddress      = "2"

**.VIFIBN2.ShimIPC[1].difName           = "Shim2ToGL1"
**.VIFIBN2.ShimIPC[1].ipcAddress        = "1"
**.VIFIBNGL1.ShimIPC[1].difName         = "Shim2ToGL1"
**.VIFIBNGL1.ShimIPC[1].ipcAddress      = "2"

**.VIFIBN24.ShimIPC[0].difName          = "Shim25To24"
**.VIFIBN24.ShimIPC[0].ipcAddress       = "1"
**.VIFIBN25.ShimIPC[0].difName          = "Shim25To24"
**.VIFIBN25.ShimIPC[0].ipcAddress       = "2"

**.VIFIBN24.ShimIPC[1].difName          = "Shim14To24"
**.VIFIBN24.ShimIPC[1].ipcAddress       = "1"
**.VIFIBN14.ShimIPC[1].difName          = "Shim14To24"
**.VIFIBN14.ShimIPC[1].ipcAddress       = "2"
```

Figure 45. Sample of simulation configuration code



```
# RMT Forwarding policies
**.VIFIBN*.TenantIPC.relayAndMux.ForwardingPolicyName = "SimpleTable"
**.VIFIBN*.GIPC.relayAndMux.ForwardingPolicyName = "SimpleTable"

**.VIFIBNGL*.IGIPC[*].relayAndMux.ForwardingPolicyName = "SimpleTable"

# forwarding generator policies
**.VIFIBN*.TenantIPC.resourceAllocator.pdufgPolicyName = "SimpleGenerator"
**.VIFIBN*.GIPC.resourceAllocator.pdufgPolicyName = "SimpleGenerator"

**.VIFIBNGL*.IGIPC[*].resourceAllocator.pdufgPolicyName = "SimpleGenerator"

# Routing policies
**.VIFIBN*.TenantIPC.routingPolicyName = "SimpleDV"
**.VIFIBN*.GIPC.routingPolicyName = "SimpleDV"

**.VIFIBNGL*.IGIPC[*].routingPolicyName = "SimpleDV"
```

Figure 46. Sample of simulation for routing and forwarding for distributed clouds use case

Simulation results

Simulations for large distributed clouds scenarios have been conducted using ad-hoc algorithms. In the following section, we present the obtained results.

Comparative of Flat DIF vs DIF divided into sub-DIFs with and without border routers

In this section we have used ad-hoc simulations. We consider three random scenarios of 210, 212 and 214 nodes. The network is managed by a single DIF. Three different methods have been compared focusing on the average and maximum distance between any pair of nodes and the size of the routing tables:

- Flat addressing (1 single sub-DIF).
- Aggregation over the same DIF (N sub-DIFs without border routers).
- Aggregation via recursion (N sub-DIFs using border routers).

For solutions consisting in dividing the DIF into N sub-DIFs, we also considered 3 distinct ratios between the number of sub-DIFs and the average number of nodes in each sub-DIF (1:4, 1:1 and 4:1). In order to get the most accurate data possible, each solution has been tested with random networks constructed with that precise solution in mind. All generated networks have been constructed with an average nodal degree of 12 (a bit less than in the current Nexedi's Distributed Cloud, where 20 tunnels are maintained in average at each node). In order to compensate for the need to pass through specific border nodes in the “aggregation via recursion” solution, in its networks, border nodes have twice the node degree, having half edges towards their sub-DIF and half towards other border nodes.

The results of these tests show a quite expectable outcome, i.e., having optimal and shortest paths when having a completely flat addressing in exchange of huge routing tables, and slightly longer paths when the solution forces aggregation of addresses, but leading to drastically reduced routing table sizes (Figure 47 shows the full results of these tests). Analyzing the two distinct solutions based on sub-DIFs, we found that forcing flows to pass through few border nodes increases the bandwidth usage at the border nodes (as all flows leaving the sub-DIF goes through one of them), and also the path length slightly in comparison to having all nodes directly knowing how to reach all sub-DIFs. However, while the ratio between the number of sub-DIFs and their size (i.e., number of nodes) does not strongly affect path length when no border routers are involved, when using border routers, more and smaller sub-DIFs even result in a better performance (both in path length and bandwidth usage, as less flows need to be aggregated).

In light of the obtained results, we would advocate the solution leveraging aggregation via recursion with these Distributed Cloud sizes. Although very slightly increased path lengths are observed, this solution makes the most of the RINA architecture recursive capabilities to come up with more manageable solutions for big networks (enabling a network manager to optimize separately the connectivity between sub-DIFs and in the Backbone DIF). Moreover, it also results in a cheaper solution to migrate into the multiple Clouds solutions, if desired.

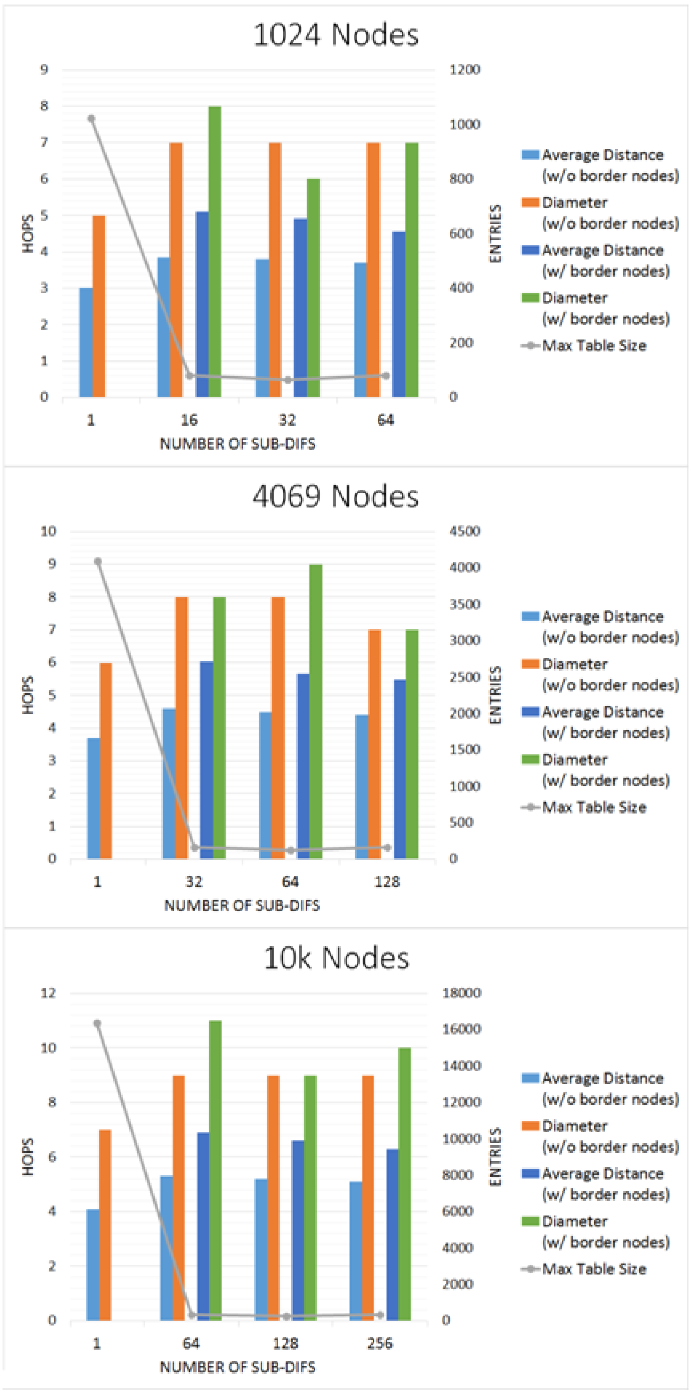


Figure 47. Comparative AVG/MAX distance in the evaluated scenarios

3.2.2. Data Centre Use Case

Characteristics and assumptions of the use case

Data Centres (DCs) are multi-tenant by nature, must support access from the outside (if providing a public cloud service) and to be distributed within multiple locations. Moreover, the support for cloud computing demands flexibility, as applications with different networking requirements are dynamically instantiated and destroyed. These characteristics make DC networking a challenge, which is very complex to meet with the current technologies:

- Multi-tenancy demands strict flow isolation, both from a security and resource allocation point of view. TCP provides poor flow isolation, as by design flows compete for the same resources, interfering with each other. Security is complicated since it is expressed in terms of IP addresses and ports, instead of application names (updating the rules is cumbersome in a changing environment, such as the DC one).
- The support of different applications with changing requirements implies the ability for the network to provide different levels of services, backed by different resource allocation techniques, which IP doesn't support. For the DC to make an efficient use of its resources and to support the high availability of applications, it is necessary to relocate running VMs to different physical machines, sometimes in another physical DC. The fact that IP doesn't easily support mobility complicates VM mobility a lot, usually restricting the movement of a VM within the same IP subnet.

Some separate solutions to the different issues have been proposed and deployed, such as DTCP to provide better flow isolation; or virtual networking to create L2 overlays on top of L3 networks (VXLAN, NVGRE, STT), thus allowing VMs to move. But all these solutions are add-ons that only address the issues partially, and further complicate the management of the DC (which in turn makes flexibility and dynamicity harder). In contrast, RINA provides a framework in which most of these problems are non-issues (access control rules are defined based on application names, congestion control and resource allocation techniques can be utilized to provide strong isolation between flows, mobility is inherently supported by the structure); therefore building upon RINA enables simpler, more efficient, easier to manage and more responsive DCs.

System Management Architecture

In order to match the Network Management scenario with the scope of the project we assume that the DC infrastructure is a single management domain, the Network Management Distributed Management System (NM-DMS). We also assume a scenario similar to the distributed clouds use case scenario in which there is logically centralized Manager Process configuring and monitoring the DC nodes via management agents deployed at each node as depicted in Figure 29. The Manager process can communicate with each ones of the agents via a separate DIF dedicated to the NM-DMS system. This DIF can run over physically or logically separated infrastructure.

Services provided by the data centre platform could be deployed using App-DAFs. App-DAF is a collection of Distributed Application Processes (DAPs) that will be sharing information using specialized tenant App-DIFs that are tailored to the needs of the App-DAFs as illustrated in Figure 30.

Addressing and Routing Policies

DIF Architecture to Bound the Routing Table Size

Hierarchical addressing schemes seem to be the ideal solution for data centres. As stated previously, in some scenarios, a data center provider may own several data centers distributed geographically. In Figure 48, we can see several distant data centres connected by the mean of network service provider facilities (internet) that is presented as the backbone.

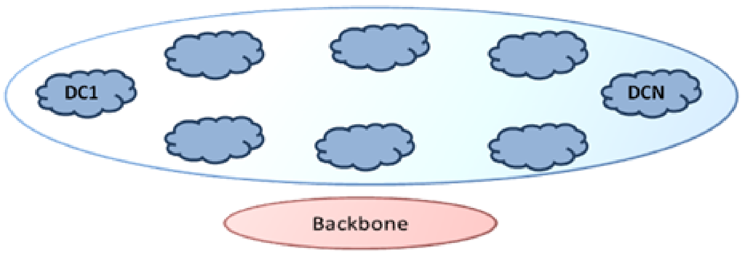


Figure 48. The RINA view of the Data Centre use case

In order to translate this view into RINA logic, a hierarchical architecture of DIFs should be deployed. As illustrated in Figure 49, basically, there will be on top a tenant App DIF managing the connection of the App-DAF. Below, a tenant DC DIF is deployed to inter-connect the several remote Data centres. This latter could regroup all the data centres of the provider. Conversely, in order to avoid managing huge DC DIF, multiple tenant DC DIFs could be set up according to the need of the provider. These DIFs are tailored to handle an efficient connectivity between the data centres, i.e. the provider may select the servers on the different data centres needing to establish connectivity and build consequently tenant DC DIFs. The inter-datacenter tenant DIF is only aware of the IPC processes on VM, servers and border routers.

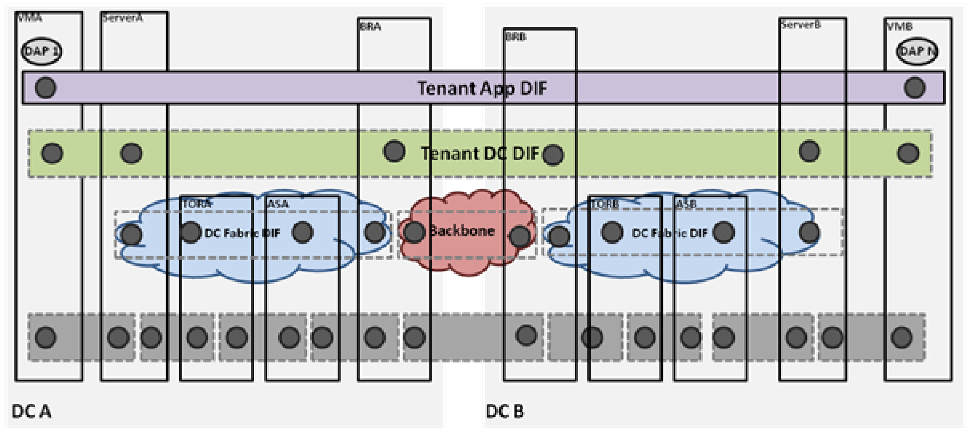


Figure 49. DIF Architecture for Data Centre use case

Then, DC Fabric DIF is intended to connect together all the hosts in the DC. It allows the provider to allocate DC resources efficiently based on the needs of its tenants. It could be managed efficiently and divided into multiple DIFs adapted to the connectivity needs. Finally, the backbone DIF is deployed to connect all the BRs in order to facilitate the routing among them and specially to interconnect the remote data centres.

Routing Policies

The routing policy that will be used in the DIFs depends basically on the topological addresses assigned to the IPC Processes. Each node in the Data Centre has to know how to reach border routers based on the address. Then, a border router can determine the next hop in the Backbone DIF based on internet connectivity. Traditional routing might be used here.

In the next section, we will tackle the address assignment issue and give a detailed example.

Topological Addressing

In Figure 50, each data center has a different identifier (from 1 to 8), i.e. this identifier is assigned to the given border router of each data center and represents the address in the backbone DIF. We assume here that all the border routers are directly connected. In the data center Fabric DIFs, IPC processes running on Top of Racks, switches and border routers are addressed hierarchically in the DC fabric DIF, following the hierarchical structure of the data center network (the addresses in blue). For instance, server with the address 1.1.1.1 is connected to the TOR 1.1.1 which in its turn connected to the AS 1.1. This latter is connected along with the AS

1.2 to the border router 1. Assuming that source node 1.1.1.1 wants to communicate with destination node 1.1.1.4, it can conclude comparing the addresses that they belong to the same data center as they have the same prefix. Therefore, they can directly communicate over the DC fabric DIF. Conversely, if the same source node wants to communicate to destination node 4.1.1.1, it will conclude by comparing the addresses that they belong to different data centers. So, communication will be performed through their border nodes in their data centers, with addresses 1 and 4 in the backbone DIF. The node 1.1.1.1 should find the route leading to the border router with the address 1. Then, it will find the route to the destination border router with address 4 (here in the example there is a direct link but we can imagine that they are connected over internet and multiple hops might be required to reach the destination).

In the inter-datacenter DIF, IPC processes on servers and border routers can also be addressed in a topological manner, starting with the identifier of the data center where they belong. An example is shown in the figure with the addresses in black.

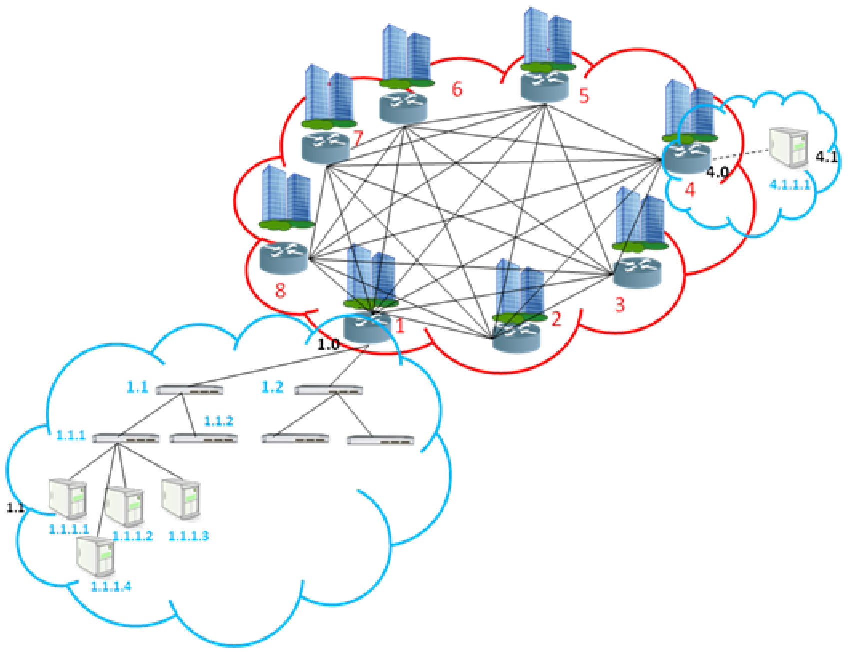


Figure 50. Topological addressing for Data Centre use case

3.3. Conclusion

In this chapter, we have detailed the generic architectures proposed for routing and addressing in PRISTINE use cases. We have presented some initial simulation results related to the distributed clouds use case. In future work, we plan to further study other PRISTINE use cases and provide more simulation results in order evaluate the assets of using RINA. Moreover, based on the evaluation results, selected architectures will be proposed for implementation in the RINA SDK.

References

[AlFares] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, “Hedera: Dynamic flow scheduling for data center networks,” in Proc. of USENIX NSDI, San Jose, CA, USA, 2010.

[Chiesa] Marco Chiesa et al, “Traffic Engineering with Equal-Cost-MultiPath: An Algorithmic perspective,” IEEE INFOCOM, 2014.

[D21] PRISTINE Consortium. Deliverable D2.1. Use cases and requirements analysis. May 2014. Available online[1].

[D22] PRISTINE Consortium. Deliverable D2.2. PRISTINE Reference Framework. June 2014. Available online[2].

[D61] PRISTINE Consortium. Deliverable D6.1. First iteration trials plan for System-level integration and validation. March 2015. Available online[3].

[Davies] N. Davies, “Delivering predictable quality in saturated networks”, Technical Report, September 2003, available online[4]

[EyeQ] EyeQ: Practical Network Performance Isolation at the Edge, imalkumar Jeyakumar, Stanford University; Mohammad Alizadeh, Stanford University and Insieme Networks; David Mazières and Balaji Prabhakar, Stanford University; Changhoon Kim and Albert Greenberg, Windows Azure, 10th USENIX Symposium on Networked Systems and Implementation.

[Greenberg] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, “Vl2: A scalable and flexible data center network,” in Proc. of ACM SIGCOMM, Barcelona, Spain, 2009.

[Guo] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, “DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers,” in Proc. of ACM SIGCOMM, Seattle, WA, USA, 2008.

[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://www.pnsol.com/public/TP-PNS-2003-09.pdf

[Hopps] C. Hopps, “Analysis of an equal-cost multi-path algorithm,” United States, 2000.

[Niranjan] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, “Portland: A scalable fault-tolerant layer 2 data center network fabric,” SIGCOMM Comput. Commun. Rev., vol. 39, no. 4, pp. 39–50, Aug. 2009. Available online[5]

[RFC2991] RFC 2991, “Multipath Issues in Unicast and Multicast Next-Hop Selection”, Nov. 2000.

[Riggio] Roberto Riggio, Francesco De Pellegrini, and Domenico Siracusa, "The Price of Virtualization: Performance Isolation in Multi–Tenants Networks", in Proc. of IEEE ManFI 2013.

[SplitTCP] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby, “Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations,” RFC 3135, June 2001.

[5] http://doi.acm.org/10.1145/1594977.1592575