University of Amsterdam
System & Network Engineering

Research Project I

Recursive InterNetwork Architecture

An Assessment of the IRATI Implementation

Jeroen van Leur *Jeroen Klomp*

Supervisors: Marijke Kaat & Ralph Koning

Sunday 7th February, 2016

Abstract

The Recursive InterNetwork Architecture (RINA) is a clean slate approach to a new internet architecture that has the promise to solve many of the long-standing problems of the current Internet. For this project we have assessed the open source RINA IRATI implementation. We have found that the implementation is still in an experimental state and that it might be difficult for external researchers to build testbeds with. Performance tests showed that the implementation has potential but routing tests showed it is still lacking in terms of resiliency.

Acknowledgements

We would like to thank our supervisors, Marijke Kaat and Ralph Koning, for providing us with valuable insights and feedback. We would also like to express our gratitude to the IRATI developers, especially Dimitri Staessens, Sander Vrijders, Eduard Grasa and Javier García, for their helpful replies to enquiries.

Contents

1 Introduction **5**

1.1 Research questions 5

1.2 Related work 6

2 Background **7**

2.1 Recursive approach of RINA 7

2.2 Networking is inter-process communication 8

2.3 Initialisation of data transfer 10

2.4 RINA Protocols 11

2.5 Addressing in RINA 12

2.6 Information management in the RINA infrastructure 14

2.7 Developments & further reading 15

3 Implementations **17**

3.1 Implementation characteristics 17

3.2 IRATI routing and configuration 19

4 Experiments **21**

4.1 Test setup 21

4.2 Scenario 1: basic tests 22

4.3 Scenario 2: routing tests 25

5 Results **29**

5.1 Scenario 1 29

5.2 Scenario 2 39

6 Discussion and recommendations **45**

6.1 Connectivity 45

6.2 Performance 45

6.3 Routing 46

6.4 Generic topics 46

6.5 Recommendations 47

7 Conclusion and future work **48**

7.1 Conclusion 48

7.2 Future work 48

Bibliography **50**

Acronyms **54**

Appendix A Test setup details **57**
A.1 KVM 57
A.2 IRATI stack 57
A.3 RINA traffic generator & Wireshark 57

Appendix B Test scenario configuration details **59**
B.1 Scenario 1: basic tests 59
B.2 Scenario 2: routing tests 65

Appendix C Workarounds for usability problems **75**
C.1 Console commands 75
C.2 Wireshark patch 76

1 Introduction

The architecture of the internet has largely been formed in the 60s, 70s and early 80s of the previous century, and while it has enabled the world-wide proliferation of communication and computing, it is plagued by long-standing architectural problems. Among these are the issues with multihoming, mobility, multicast, scalability, quality of service and security.

The Recursive InterNetwork Architecture, as envisioned by John Day, is a clean-slate approach that has the potential to address the problems and to enable new innovative networking applications. It revolves around the vision that networking basically is interprocess communication and that a network model is not a series of distinct interchangeable layers but consists of a single repeatable and programmable layer where mechanism and policy are separated.[1]

After the initial publication of John Day in 2008 several research initiatives were formed to study this new model and create experimental implementations. To coordinate these initiatives the Pouzin Society[1] was founded.[2] According to the research projects as overlooked by the Pouzin Society there are currently three implementations that are actively being worked on: protoRINA, OpenIRATI and RINASim.[3]

The RINA model has not been fully defined yet and these implementations are thus incomplete and experimental. Furthermore the mentioned prototypes are mainly tested by their developers, therefore we would like to know what the current state is of (one of) the implementations and whether it can be used by independent researchers to conduct their own tests.

1.1 Research questions

The purpose of the research is to get a clear overview of RINA and how it is currently implemented. The reason IRATI was chosen is because of its open source nature and active development. It also appears to be the most advanced implementation in terms of its resemblance to a native stack. During the project emphasis is laid upon the routing aspects of IRATI.

The main research question is: *What is the current state of the IRATI RINA implementation?*

The main question can be split into the following sub-questions:

- Which other implementations are there and how do they relate to IRATI?
- To what extent is the RINA model implemented in terms of routing?
- How resilient is the routing to link failure?

[1] Named after Louis Pouzin, the inventor of the datagram.

1.2 Related work

There are several groups that do research on RINA.[3] In the following list the most relevant projects and papers are given.

- ProtoRINA; led by John Day, the RINA research team works on investigating the fundamentals of RINA, and has developed an open source implementation called ProtoRINA. It is programmed in Java and runs over User Datagram Protocol (UDP). The implementation has been tested on the GENI infrastructure and while the development of the prototype does not seem to be very active the project has not been finished officially.[4]–[8]
- IRATI; the now finished FP7 IRATI project has resulted in the OpenIRATI implementation, an open source RINA implementation built on top of Linux and uses Ethernet. FP7 is a programme launched by the European Union that aims to enhance Research and Technological Development. After the project was finished the development still continues e.g., by other research projects. The research group consisted of five teams, of which one was Boston University.[9], [10]
- IRINA; with a team of four research groups, IRINA researched RINA as the foundation for modern NREN and GÉANT network architectures. The IRATI implementation was used during the study. The project is finished and has mainly contributed testing tools to the IRATI prototype.[11]
- PRISTINE; a large research group that was set up to "explore the programmability aspects of RINA to implement innovative policies for congestion control, resource allocation, routing, security and network management".[3] During the project RINASim was built, a RINA simulator within OMNeT++. The project runs until the middle of 2016.[12], [13]
- ARCFIRE; an upcoming project that aims to do large-scale RINA benchmarks in the FIRE+ infrastructure.[14]

Besides these projects several papers were published about the routing aspects:

- In *Bounding the router table size in an ISP network using RINA* John Day et al. present RINA as a solution to the current scalability problems of the internet and emphasise the purpose of the improvements to hierarchical addressing in terms of routing efficiency.[15]
- In *Network Discovery in a Recursive Internet Network Architecture* Sarina Knappstein-Hamelink investigates different Distributed IPC Facility (DIF) discovery methods, routing algorithms and path finding strategies.[16]

2 Background

In this chapter the general ideas behind Recursive InterNetwork Architecture and the main components it consists of are introduced.

2.1 Recursive approach of RINA

The model and main set of communication protocols the Internet is built upon, is called the Internet protocol suite (TCP/IP).[17] This model is divided into several distinct layers in order to hide the complexity behind abstractions (and often is to be believed to modularise the model). Every layer provides a specific service to the layer immediately above it.

The problem with these layers is that they are not well-defined and do not provide enough flexibility; the same functionality is offered across different scopes, the provided services of the layers violate the scope of the layer or the provided services are incomplete (e.g., reliability is offered in the data link layer but also in the transmission layer, flow control provisions in the transport layer might not be suitable to certain physical media, the network links have multiple addresses, IP addresses name the network interface controller (NIC) instead of the node, domain names are merely macros for IP addresses, application port numbers are again a macro for an application type, nodes and applications do not have addresses or names, etc.).

This incomplete network architecture and its inconsistencies lead to many problems that require constant workarounds. These workarounds introduce unnecessary complexity and inefficiency because every use case needs a different solution e.g., in the form of a specific protocol. (For a complete analysis of the shortcomings of TCP/IP see the book *Patterns in Network Architecture* by John Day.)

In RINA all services are provided in one layer. The layers are set up recursively which means that the same protocols can be repeated in a protocol stack, encapsulating each layer in another of itself.[18] In contrast to TCP/IP the amount of layers that exist in a system is not fixed. Instead the number of layers is variable and the actual amount depends on what is required for the communication.

Whereas in TCP/IP the layers are called by their name or a fixed number (e.g., the Internet layer or Layer 3 of the OSI model), in RINA this needs to be accommodated via a different construction due to its recursive nature. The convention is to take the viewpoint of the current layer and call it the (N)-layer. The layers above and below it are respectively called the (N+1) and (N-1)-layer. When the depth of the layers is known the variables can be replaced by discrete numbers; the lowest layer is called layer 1, the one directly above it layer 2 and so forth. Ultimately the lowest layer is always the physical layer.

It is not important how many layers there are, each acts independently of the others. This however does not mean that data does not flow through other layers. To communicate with applications on other systems the flows need to traverse through the different layers, the difference with the TCP/IP model is that the starting and ending levels are not fixed but depend on what level the applications are connected to. However, two applications need to be connected to the same layer in order to be able communicate with each other.

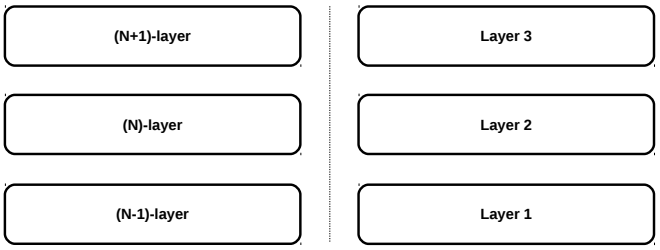

Figure 2.1: Layers in RINA. On the left-side the layers are shown from the viewpoint of the middle layer while the total amount is unknown or irrelevant. The right-side shows numbered layers in an environment where the amount is known.

2.2 Networking is inter-process communication

RINA is built on the premise[1] that *"networking is not a layered set of different functions but rather a single layer of distributed inter-process communication that repeats over different scopes. Each instance of this repeating IPC layer implements the same functions / mechanisms but policies are tuned to operate over different ranges of the performance space (e.g., capacity, delay, loss)."*[19, p. 1]

This concerns the general inter-process communication (IPC) model, i.e. two applications have a means to share data and state in a distributed fashion, but not a specific implementation. What this means in RINA is that all communication is facilitated by underlying processes which are similar to the applications that are themselves carrying out IPC. The difference is that end-user applications are general purpose while the underlying processes are specifically providing an IPC facility.

To understand RINA there are a couple of definitions that one needs to get acquainted with. We have selected the most important ones (the complete reference can be found in John Day's book)[1]:

- *Application Process (AP); a networked application. Consists of one or more Application Entities (AEs), of which there may be multiple instances of each.*
- *Application Entity (AE); instantiations of the application protocols. An application process can have not only different kinds of AEs, but also multiple instances of them, and there could be multiple instances of the AP in the same system.*
- *Distributed Application Process (DAP); the instantiation of a computer program executing in a processing system intended to accomplish some purpose. A DAP contains one or more tasks or AEs, as well as functions for managing the resources (processor, storage, and IPC) allocated to this DAP. A DAP is a member of a Distributed Application Facility (DAF).*
- *Distributed Application Facility (DAF); a collection of two or more cooperating APs in one or more processing systems, which exchange information using IPC and maintain shared state.*

[1]Based on Robert Metcalfe's quote from 1972.

- *Distributed IPC Facility (DIF); a collection of two or more DAPs cooperating to provide IPC. A DIF is a DAF that does only IPC. The DIF provides IPC services to APs via a set of API primitives that are used to exchange information with the application's peer. In most cases, the inter-process communication process (IPCPs) comprising the DIF are in different systems. In terms of TCP/IP a network can be seen as a DIF in RINA.*[2]

- *Inter-process Communication Process (IPCP); a special AP within a DIF delivering IPC. An IPCP is an instantiation of a DIF membership. An IPCP is a specialized DAP. In RINA IPCPs are equivalent to the nodes*[2] *of TCP/IP and since a system can have multiple DIFs multiple of these IPCPs can reside on a single system, or even within the same DIF on a single system.*

- *Processing system; the hardware and software capable of executing programs instantiated as DAPs that can coordinate with the equivalent of a "test and set" instruction, i.e. the tasks can all atomically reference the same memory. In TCP/IP this is referred to as a node.*[2]

Figure 2.2 shows an overview of the RINA architecture. There are three levels of DIFs; the lowest are of smallest scope comprising the physical media while the ones above are larger in scope and ultimately provide services to the DAF at the top where the APs are located. In this example there are two APs; one named S (for source) and named D (for destination). Both are registered to DIF A (the APs make themselves known to the DIF by registering in the directory service provided by that DIF). All the IPCPs (or nodes)[2] that are enrolled to (members of) DIF A are called neighbours if they share an (N-1)-DIF (in RINA terms a link)[2].

In RINA there are three kind of systems: hosts, border routers and interior (or sometimes called internal) routers. Vladimír Veselý, describes these as follows[21, p. 2]:

a. *"hosts; IPC end-devices containing AEs, they employ two or more DIF levels;*

b. *interior routers; interim devices interconnecting (N)-DIF neighbors via multiple (N-1)-DIFs, they employ two or more DIF levels;*

c. *border routers; interim devices interconnecting (N)-DIF neighbors via (N-1)-DIFs, where some of (N-1)-DIFs are reachable only through (N-2)-DIFs, they employ three or more DIF levels."*

[2]When adhering to the terminology as defined by IRATI.[20] However, RINA is still in development and there is not a definite official reference yet. Therefore the terminology might be a bit confusing or turn out to be wrong and could be susceptible to change in the future. To avoid confusion we will try to refer to specific RINA components where possible.

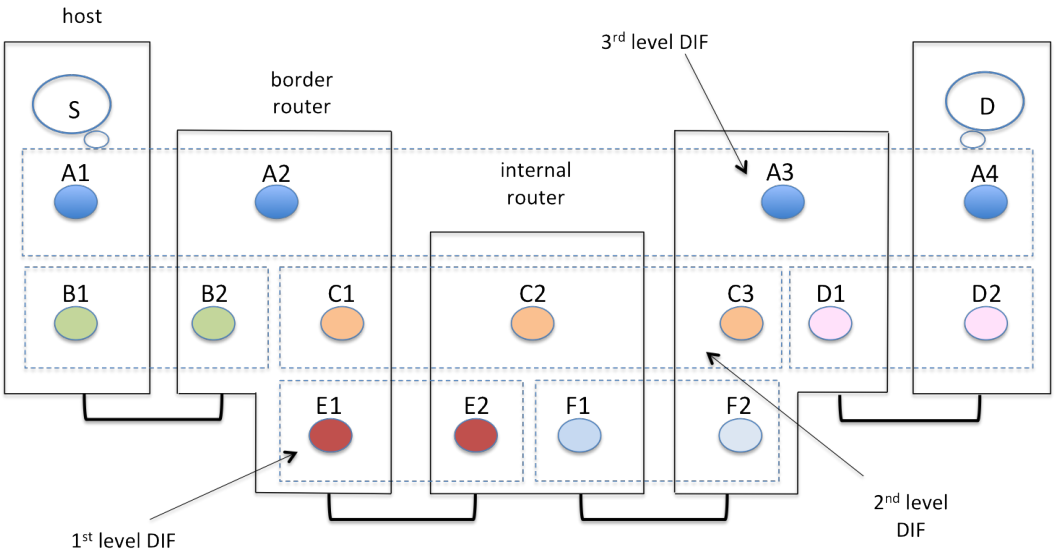

Figure 2.2: The RINA architecure[3]

2.3 Initialisation of data transfer

The IPCPs function as the gateways for the APs; they facilitate in IPC by creating a DIF via collaboration with other IPCPs. Before data can be transferred over a DIF, every IPCP has to undergo the following phases:

1. Enrolment Phase
2. Allocation Phase
3. Data Transfer Phase

The *Enrolment Phase* consists of connecting an IPCP with an existing DIF. The existing DIF can be found using a DIF discovery mechanism, or by creating a new DIF when there is no suitable DIF available. Enrolling creates, maintains, distributes and ultimately deletes information within a DIF. Such information can be addressing, access-control rules or other management policies required to create and characterise communication. Enrolling results in IPCPs to discover each other and then become neighbours in a DIF.

The *Allocation Phase* creates, maintains and deletes the information required to support the function of the data transfer phase for a particular IPCP [1]. This can consist of binding the IPCP to the N-1 layer. The allocation phase is performed by the Flow Allocator. Flow allocation is responsible for creating and managing an instance of IPC, also known as a flow.

The last phase is the *Data Transfer Phase* where the actual transfer of data occurs.

[3]Source of the figure: http://irati.eu/the-recursive-internetwork-architecture/

2.4 RINA Protocols

To define the semantics of the data flows RINA uses protocols, just like TCP/IP. Whereas TCP/IP has a huge amount of protocols, RINA only requires two protocols[22]. These are the Common Distributed Application Protocol (CDAP) and the Error and Flow Control Protocol (EFCP).

2.4.1 Error and Flow Control Protocol

The Error and Flow Control Protocol (EFCP) provides both unreliable and reliable mechanisms for communication. It includes Data Transfer Protocol (DTP) for the transfer of service data units (SDUs) (payloads), and functions like fragmentation, reassembly, sequencing, addressing, etc. DTP has some resemblance to the User Datagram Protocol due to the absence of control and flow mechanisms and because it does not keep state.

The Data Transfer Control Protocol (DTCP) is used in order to provide flow control, transmission control and retransmission control. This protocol maintains state and operates separately from DTP. Whether Data Transfer Control Protocol (DTCP) is used is a matter of policy (i.e. within DIFs it is a choice to enable DTCP, its characteristics are configurable and applications can select a policy that suits their requirements).

EFCP is based on the Watson's delta-t transport protocol.[23] The protocol provides the necessary synchronisation between two IPCPs using three local timers. The three timers are MPL, A, and R [24]. Watson proved that distributed synchronisation can be reached by only using these three timers:

- maximum packet lifetime (MPL): the upper bound that a packet can exist in a network
- maximum number of retries (R): the maximum time a sender will keep retransmitting a packet
- maximum time before ACK (A): the maximum time a receiver will wait before sending an ACK

Via these timers there is no need for explicit synchronization (as in Transmission Control Protocol (TCP)) thus connections do not need to be set-up prior to the sending of data nor need they be explicitly destroyed. The minimum delta time for senders is $3(MPL + R + A)$ while receivers need to have a delta time of $2(MPL + R + A)$. After the timer reaches zero all state can be discarded. In Listing 1 the formula of the delta-t protocol is given for clarity.

$$\Delta t = MPL + R + A$$

Listing 1: Formula delta-t protocol

2.4.2 Common Distributed Application Protocol

The Common Distributed Application Protocol (CDAP) is used by APs to exchange any structured application-specific information and is equivalent to an application protocol of the TCP/IP model. However, in RINA there is only one application protocol; Common Distributed Application Protocol (CDAP), which can be used to by programmers to carry application-specific data structures. CDAP provides six primitive operations to accommodate all the possible use cases: create, delete, read, write, start and stop. Besides by end-user APs CDAP is also used by IPCPs (within DIFs) e.g., in order to exchange DIF management information.

2.5 Addressing in RINA

In TCP/IP a connection is established between nodes by coupling two sockets (Internet Protocol (IP) addresses and port numbers). An IP address is a logical address that does not name the node but names an interface, which is a source of certain problems of the Internet (e.g., multihoming and mobility). On a node applications are identified only by port number, they do not have names within the architecture.

In RINA however,[4] connections are made between APs and are requested by name. This name is a string of a variable length and is unambiguous within the architecture (i.e. globally unique). Application names are location-independent, which facilitates mobility.

Within DIFs IPCPs are assigned addresses which could be leveraged to improve routing and DIF management. Addresses are synonyms for fully qualified IPCP instances and are only unambiguous within a DIF. One reason that addresses are used within DIFs is that they are location-dependent, which can make routing more efficient. The length of the address may vary, which could be based on the size of the DIF. If the DIF has a large topological coverage this address might be relatively long. Since every network consists of DIFs that cover a different scope, an ISP with a larger scope may use addresses of a larger size.

When APs request a communication channel with another program they are assigned a port-id. A port-id is the same concept as a file descriptor and is used to distinguish between multiple flows that an AP partakes in. The port-id is unambiguous per AP and DIF. When initiating a flow APs need not know any other information about the other AP than the name of the remote AP and the supplied local port-id. When data need to be transferred the AP can simply use the port-id to send it and the DIF will be responsible for the transmission.

To be able to carry out its task the IPCP the AP is connected to will allocate a connection-endpoint identifier (CEP-id) per requested flow[5] and pair it with the AP's port-id. Then when data is sent by an AP the IPCP will create a connection[5] with a remote IPCP over which they will exchange protocol data units (PDUs). The IPCP's PDUs contain protocol control information (PCI) like it's own address as the source, the destination address of the IPCP the remote AP is connected to. Furthermore a connection-identifier is constructed by appending the source's and remote's CEP-id plus the quality of service (QoS)-id. The connection-identifier is unambiguous between two IPCPs (to be precise two Error and Flow Control Protocol Machines (EFCPMs), which is an instantiation of the EFCP) in a single

[4] Which is structured according to Jerry Saltzer's addressing architecture published in 1982 that was republished in 1993 as RFC 1498.[25]

[5] In RINA terms a flow is used for end-to-end channels between APs, while connections are used for channels between EFCP instances.

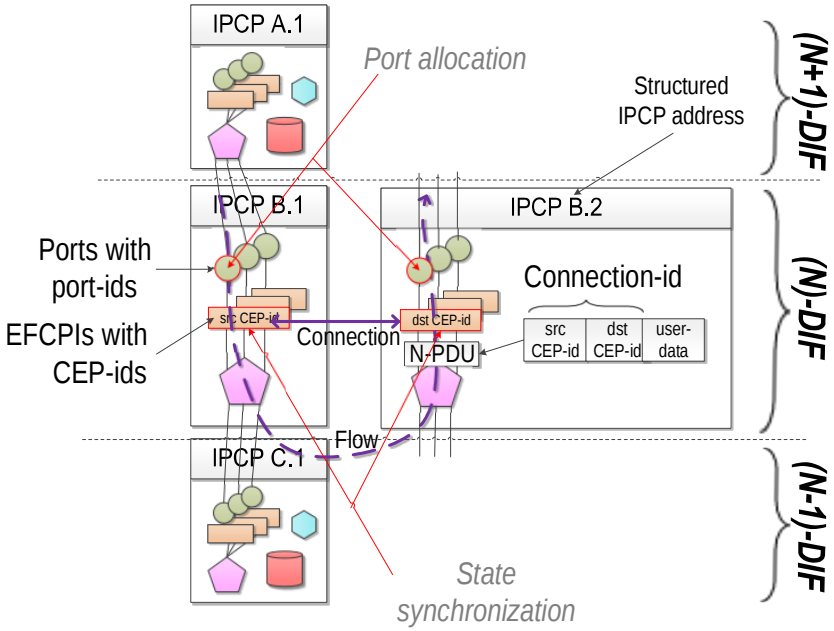

Figure 2.3: IPCP's local identifiers overview [12]

DIF. When necessary the connection-ids can be rolled over. In Figure 2.3 the IPCPs local identifiers are shown.

There is one more name that is crucial for the architecture to work; the point of attachment. From the point of view of the (N)-level IPCP the directly connected IPCP one level below it is called its point of attachment. The other way around is that the lower IPCP sees the one above it as an AP. Routing works by finding a route in the same DIF to the final IPCP that is connected to an AP. Then when the next hop is found a path to it via a lower lying IPCP is selected. This is repeated until the packet reaches its destination. Point of attachments are unambiguous between adjacent nodes and path-dependent. The relation between AP names, node addresses, point of attachments, routes and paths is shown in Figure 2.4 while in Table 2.1 an overview of the names required for any well-formed network architecture are given.

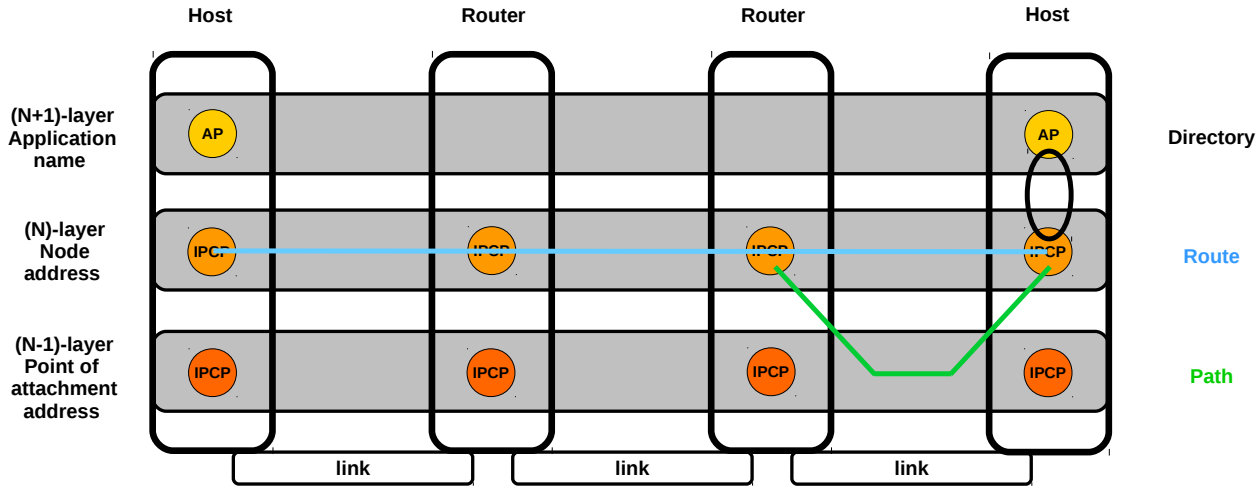

Figure 2.4: Directory, routes and paths in RINA (based on [26])

Common Term	Scope	Application Term
Application Process Name	Global (unambiguous)	Application Process Name
Address	DIF (unambiguous)	Synonym for an IPC Process Name
Port-id	Allocation AE of an IPC Process (unique)	AE-instance-identifier
Connection-endpoint-id	Data Transfer AE of an IPC Process (unique)	AE-instance-identifier
Connection-id	Src/Dest Data Transfer AEs (unique)	Concatenation of data transfer AE-instance-identifiers
DIF Management Exchange	IPC Process (unambiguous)	AE-identifier

Table 2.1: Summary of names [27]

2.6 Information management in the RINA infrastructure

In RINA the state of applications in regards to network communication is stored in the resource information base (RIB). This also means that all the information necessary for the functioning of a DIF is stored in the RIB (e.g., mappings of addresses, registered applications, established flows forwarding and routing tables).[20, p. 19]

In order to facilitate discovery of applications and DIFs an Inter-DIF Directory task is used. The Inter-DIF Directory (IDD) is a DAF that helps with finding and allocating the correct DIF in order to setup the data flow between applications. The IDD will be

configurable with specific search rules so that it can find applications outside of its own scope.

There are various of other important components which have not been discussed yet, examples are:

- Relaying and Multiplexing Task: the relaying task is used for forwarding PDUs between different IPCPs. Routing decisions are based on routing information and QoS. The multiplexing task is used for sending data to lower (N-1)-DIFs via its point of attachment.
- SDU Protection: provides protection of SDUs (e.g., error detection, time to live, encryption and compression).
- Management Agent: used for management of the DIFs. Via the management agent administrators can change policies (like the configuration of authentication modules and QoS).

An overview of the RINA reference model is shown in Figure 2.5.

2.7 Developments & further reading

As explained in its current form RINA would not scale very well (e.g., since DIFs are distributed and we have not specified how information propagates or is kept consistent, RIBs would need to be globally the same in order for this model to function properly). Therefore there are several optimizations made, proposed and/or hypothesised that would allow RINA to scale to a similar size as the internet. Techniques such as subnets within DIFs, topological namespaces, topological routing, and topological directories are part of the research topics as well as novel routing algorithms that fit the recursive model.

The RINA model does not have a definite specification yet. There are however a lot of papers and documentation about the reference model and the implementations. According to the IRATI documentation there is an official RINA Specification Handbook published by the Pouzin Society on January 2013. However, information about this handbook is sparse and it does not appear to be available online. As part of the ProtoRINA project a RINA reference model was worked upon but also in this aspect information is sparse. On the RINA Specs Wiki, which seems to originate from RINASim, several specification-like documents are available. These documents seem to originate from the before mentioned ProtoRINA project and their dates and markings (e.g., draft and confidential) appear to indicate that these are working versions of the conceptual specification.[28] After consulting the IRATI mailing list it was confirmed that the Pouzin Society is working on the specification in private and that aforementioned documents are indeed (older) parts of the reference model.

To learn more about RINA a good starting point would be to consult the Rina Specs Wiki (although the documents may be outdated), the IRATI documentation, for instance the RINA specifications and high-level software architecture,[20] and the PRISTINE documentation.[29], [30] John Day's book is also a good source to learn about the design of the architecture (and the reasoning behind it).[1]

[6]Source of figure: http://ict-pristine.eu/wp-content/uploads/2014/11/d22-rina-arch3.png

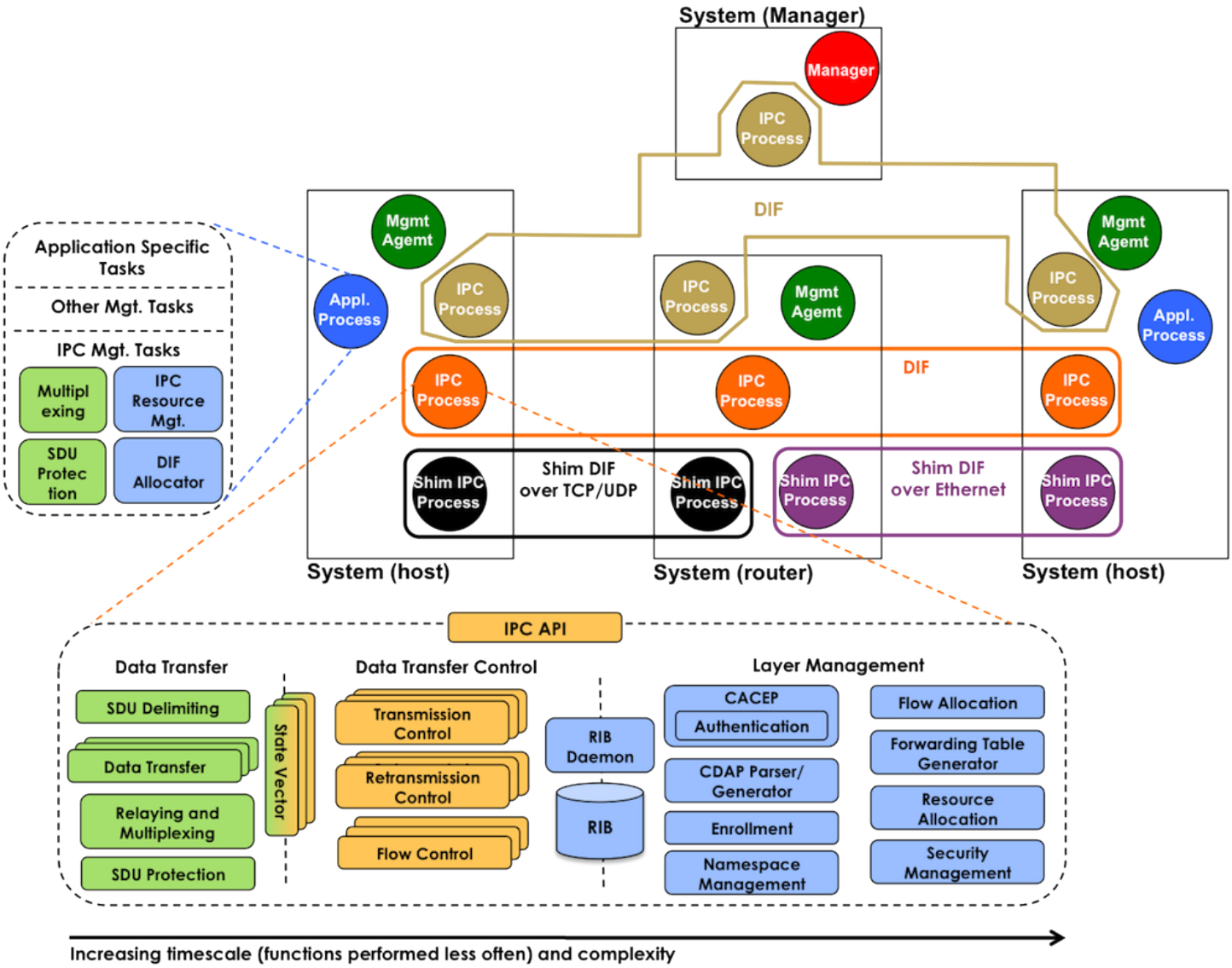

Figure 2.5: The RINA reference model[6]

3 Implementations

As stated in the introduction (see Section 1.2) there are or have been several initiatives that have studied RINA which has resulted in several experimental RINA implementations. In this chapter we look at some of the implementations and then focus on the one that is central to this project; IRATI.

3.1 Implementation characteristics

3.1.1 ProtoRINA

An important RINA implementation was created by the RINA team (led by John Day) of the Boston University Computer Science Department. Version 1.0 was released in October 2013 and is freely available online. It was created *"as a vehicle for making RINA concepts concrete and for encouraging researchers to use and benefit from the prototype"*.[22, p. 1]

ProtoRINA is written in JAVA and should be able to run on any platform that is Java-enabled. The Pouzin Society projects page states that the prototype functions over UDP/IP,[3] although the manual does not mention this.[7] According to the introductory paper ProtoRINA can leverage shim DIFs which enables it to function over Ethernet, TCP and UDP.[22, p. 2]

3.1.2 Alba/TINOS

The IRATI website lists ALBA as a RINA implementation on top of the TINOS protocol experimentation framework.[31] The origin of the name, ALBA, is unclear but the prototype was *"mainly a tool for research and experimentation, the outcome of this work is not meant to be the final solution, but a vehicle to think deeply and experiment around. RINA open research areas"*.[4, p. 1] It was developed in 2012 and its goal appears to be similar to those of ProtoRINA. Contrary to ProtoRINA, Alba does not appear to be actively developed any longer (i.e. the RINA patches to TINOS have been reverted).[32]

3.1.3 TRIA

As with the Alba/TINOS implementation, there is not a lot of background information about TRIA's prototype. What is know is that around 2011 the American startup TRIA Network Systems collaborated with *"the Irish research centre TSSG"* and *"the Boston University [...] to develop the first RINA prototype"*.[33] How this collaboration worked out is unclear, but TRIA seems to have created a close source prototype in C and C++ with the goal to *"move into the OS kernel (Linux/UNIX/etc.) eventually"*.[24, p. 22],[34] TRIA Network Systems, LLC still appears to be actively researching RINA, judging by its assigned patents.[35]–[38]

3.1.4 RINASim

RINASim was created in the frame of the PRISTINE project. The goal of the implementation was to provide a full-fledged RINA simulator in order to support ongoing research and academic activities.[21] The simulator is for the OMNeT++ simulator and describes the RINA components and basic order of operations. One outcome of the PRISTINE project was the enhancement of the IRATI implementation. It is unknown whether RINASim, developed within the same project, is based on the IRATI implementation.

3.1.5 IRATI

Between 2013 and 2014 the research project FP7 IRATI[1] was run. Its goal was to create a high performance stack that was integrated into an actual operating system. The project started as a closed source project but transitioned to an open source model and is now available via a GitHub repository[39]. Probably due to this transitions and its integration with GNU/Linux it is sometimes referred to as OpenIRATI.

The IRATI stack is divided into kernel (fast path) and user-space (slow path) components in order to provide both performance and flexibility. The implementation should be functional on all GNU/Linux OS platforms.[40] The prototype was planned to be built for the FreeBSD operating system as well and possibly also built upon Juniper's JunOS (which is based on FreeBSD). However, the current status of the release of the FreeBSD version is unknown. IRATI is planned to be improved in the period of 2014-2016, the project PRISTINE[2] is using IRATI's result to enhance its possibilities which consists of an SDK (software development kit), DIF Management System and Sophisticated policies.

IRATI supports several shim DIFs for legacy and experimental support: shim DIF over TCP/UDP, shim DIF over 802.1Q (Ethernet), dummy shim DIF which acts like a local loopback DIF used for debugging, hypervisor and host communication shim DIF. IRATI comes with several tools that can test the network infrastructure. These tools are *rina-echo-time, rina-cdap-echo, rina-tgen* and *wireshark dissectors* which are in alpha phase. The implementation also supports authentication to control access of a DIF by using RSA keys (SSH authentication) and SDU protection which allows all traffic to be encrypted using AES with a 128-bit key.

[1] consisting of five partners: i2CAT, Nextworks, iMinds, Interoute and Boston University.[3]

[2] consisting of fifteen partners: WIT-TSSG, i2CAT, Nextworks, Telefonica I+D, Thales, Nexedi, B-ISDN, Atos, University of Oslo, Juniper Networks, Brno University, IMT-TSP, CREATE-NET, iMinds and UPC.

3.1.6 Implementation overview

In the following table an overview of the RINA implementations is given (see Table 3.1).

Implementation	Organization/project	Characteristics
ProtoRINA	Boston University	written in JAVA
		runs over TCP via a shim DIF
		has only basic support for EFCP (only DTP not DTCP)
Alba/TINOS	i2CAT & TSSG	Java
TRIA	TRIA Network Systems	C & C++
RINASim	FP7 PRISTINE	RINA simulator built on OMNeT++
IRATI	FP7 IRATI	kernel components written in C; user-space components written in C++; JAVA bindings available
		supports shim DIF for UDP/IP, shim DIF for hypervisors, shim DIF over 8021Q and shim DIF for debugging purposes
		supports SSH authentication with RSA keys and SDU protection using AES-128

Table 3.1: Overview of RINA implementations

3.2 IRATI routing and configuration

The IRATI implementation requires every system in the network to be configured according to pre-defined settings prior to launch. The IPCPs that are created have to be initialised as well as the DIFs that they will be assigned to and/or registered at. The current version of IRATI does not support dynamic address assignment, therefore every system must be configured with the addresses of all IPCPs.

IRATI can run on top of other networks via shim DIFs. These shim DIFs often do not support the full RINA features, like authentication and QoS. For example in case of the virtual local area network (VLAN) shim DIF it is not possible to register normal applications in it nor does it allow for registering multiple IPCPs. Among other reasons IRATI uses the `0xD1F0` Ethertype to facilitate debugging.[20, p. 54]

When starting the connection the shim DIFs are automatically connected with each other. However to enable an application to send data over the RINA network a normal DIF must be started. For this the user must manually enrol the remote IPCP to the local DIF to set-up the layered network. Once all the IPCPs are enrolled to the DIF applications can make use of the IPCPs and send data to applications on other systems.

According to Dimitri Staessens, one of the IRATI developers,[3] the current prototype only has basic support for *link-state* routing, which is based on IS-IS in IP,[41] and fast

[3]During the project we consulted several IRATI developers via Internet Relay Chat (IRC) or e-mail.

recovery using Loop-Free Alternates approach.[42] IS-IS uses the Dijkstra's algorithm to compute the best path through the network.[43]. These technologies should enable the IRATI stack to quickly recover from link failures. In theory recovery should be within 50 ms and according to a test held during the Pristine project this is a realistic estimate (during the experiment a target recovery duration of 100 ms was set but recovery only took 1.7 ms in certain occasions).[30, p. 29]

In December 2015 a plugin for equal-cost multipath routing (ECMP) written by Javier García was merged.[44] ECMP is a routing mechanism which can route packets in parallel along multiple equal cost paths. Among others it can be used to increase bandwidth and reliability.

As stated in Chapter 1 the RINA model has not been fully defined yet and these implementations are thus incomplete and experimental. Examples of aspects that are missing are topological addressing schemes and scalable routing protocols. Furthermore important mechanisms that would allow RINA to scale are missing or being worked upon e.g., the IDD (used for discovery of remote applications and DIFs) and the namespace manager (a component used to coordinate the way applications are named). During the IRATI and PRISTINE some of the aspects of these components were specified. In the documentation of the projects elaborate explanations of the functioning of routing in IRATI are also given.[29, pp. 65,88]

4 Experiments

This chapter explains the way the experiments are conducted. First the virtualised test setup is specified, then the different test scenarios are detailed. We also explain the test cases that are executed and the expected results from the test cases.

4.1 Test setup

In order to conduct our experiments we made a virtualised testbed. We used the KVM hypervisor because of its open source nature and its high performance network capabilities (via VirtIO). The IRATI stack was installed on Debian 8.2 (Jessie) because at the time of the experiments this is the up to date version which has all the required packages in its repositories (see Table 4.2 for details on the used programs). Because of the small scale and low performance requirements of the testbed – due to the focus on routing, instead of e.g. bandwidth – the virtualised environment was built on general-purpose desktops (see Table 4.1). We decided to only make use of the Ethernet shim DIFs due to the monitoring and network configuration possibilities, instead of also testing the hypervisor (HV) and/or TCP/IP connection methods (the hypervisor shim DIF requires a patched hypervisor which could introduce instability and the TCP/IP shim DIF does not offer advantages over Ethernet during our experiments).

Component	**Specification**
System	Dell Inc. OptiPlex 7010
CPU	Intel Core i5-3570S
RAM	8 GB

Table 4.1: Hardware used for experiments

Component	**Version**
Host operation system	Ubuntu 15.10
Guest operation system	Debian 8.2
Virtualisation packages	libvirt-bin 1.2.16
	qemu 1:2.3
IRATI stack	master 9d796f2
	pristine-1.4 9ce9644
IRATI Traffic Generator	6aa1127[1]
IRATI Wireshark	c52c0e2

Table 4.2: Software used for experiments

4.2 Scenario 1: basic tests

In order to get acquainted with the IRATI stack we first tested a small two-node network. This setup is based on the first tutorial from the IRATI github wiki.[45] It consists of two nodes that are connected via a shim DIF (Ethernet connection) with one normal DIF (a non shim DIF) on top of it. We kept the configuration close to the defaults to make it easier to reproduce the results (after adjusting the configuration of the tutorial to fit the required configuration of the updated stack we found that the only difference between the first tutorial and the default configuration is the VLAN-ID). This results in that no changes need to be made to the default configuration, only the interfaces need to be brought up and the remote IPCP enrolled to the local DIF. In Figures 4.1 and 4.2 visual representations of the setup are shown.

4.2.1 Test cases

The following tests were conducted:

- Launching the IRATI stack and enrolling the remote system to the local DIF.
- Connectivity test using IRATI rina-echo-time (similar to ping). The default options will be used (e.g. the ping type is tested instead of the perf or flood types).
- Performance test using IRATI rina-tgen (RINA traffic generator, similar to iperf). Several SDU sizes will be tested (from 500 to 64,000 with increments of factor three).

We conducted these tests to make sure the stack was functioning correctly. The rina-tgen program was run in order to compare the performance from the test setup to the experiments that were done during the Pristine project. To have a baseline of speed potential the results were compared with the TCP/IP stack by running iperf. These bandwidth tests were merely done to test our installation, not to achieve maximum possible goodput. However, we did use previous results to compare our results with.[11, p. 10] During these tests Wireshark for IRATI featuring the CDAP and EFCP dissectors was tried out on the bridge to determine whether Wireshark is a good method to get insight into the flows and packets. We also inspected the RIBs during the test in order to get insight in the way routing is done.

[1]Initially the master branch was used, but it seemed unstable. Using the GitHub network graph feature we identified the most current branch, which seemed to be more stable. At first the kernel modules were compiled using their default options but after initial testing performance and stability issues arose. (See Section 4.3.4, Chapter 5, and Appendix A for more information.)

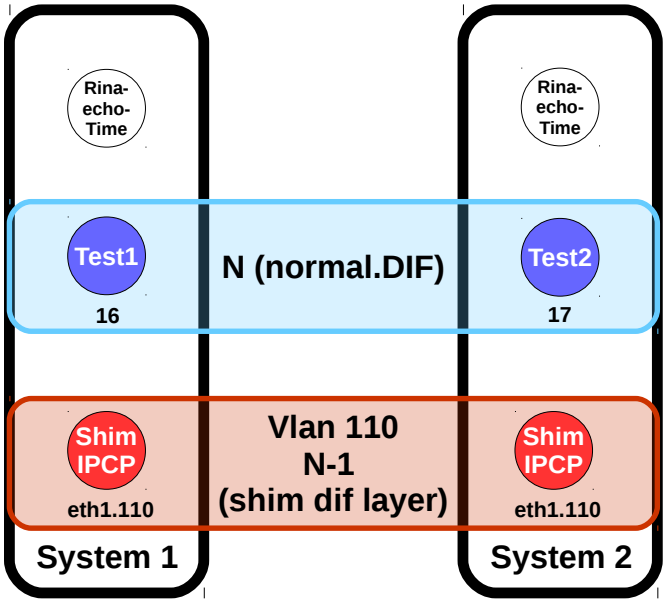

Figure 4.1: Logical design scenario 1

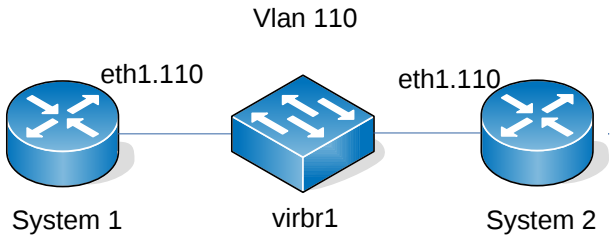

Figure 4.2: Physical design scenario 1

4.2.2 Expected results

We expected the tests to have the following outcomes:

- Since this setup resembles the first tutorial we expected no problems during the setup, start-up of the stack and connectivity tests.
- Because we use virtual machines (VMs) where the network stays local we expected the results to show low latencies (under or around one millisecond).
- In similar fashion we expected the performance to be similar to those shown in the final report of the IRINA project. Since our stand-alone hardware is most likely less capable than their virtualised environment, the results could be somewhat lower.[11, p. 10]

4.2.3 Encountered problems

During the experiments we encountered the following problems:

- When we attempted to reproduce the first tutorial launch errors and in some cases even kernel crashes occurred. This seemed to be caused by the first tutorial being outdated and the inter-process communication manager (IPCM) improperly checking the configuration files for outdated and invalid directives. After checking the GitHub repositories we found that there were more recent and more active branches. We chose to use one of the PRISTINE branches because the last stable release was from the 8th of October, the master branch had an older kernel (and we were experiencing kernel panics), the master branch did not seem to get updated and the multipath plugin was only available in the PRISTINE branches. The newer branch still could not work with the older configuration files of the first tutorial, but seemed to handle misconfiguration a bit better, so we adapted it to match the new requirements. The first scenario is the result of these findings and is used as the starting point of the experiments.
- Problems could still arise when making configuration errors (e.g., failing to bring up the VLAN prior to launching the IPCM) or when making an error during runtime (e.g., enrolling the wrong IPCP to the wrong DIF or enrolling an IPCP multiple times). This could lead to inconsistencies like an unresponsive IPCM console or the inability to enrol other IPCPs. In some occasions normal operation could be restored by bringing down the IPCM and launching it again, but at other times it was required to restart the VM. This led to the standard procedure that we would restart all the VMs if inconsistent behaviour was observed or likely to be expected.
- The performance tests resulted in abysmal throughput. It also resulted in high CPU load and a full file system. The high CPU load was caused by busy syslog processes and the full file system was caused by gigabytes of logging information. This was caused by the debugging options in the IRATI kernel modules which resulted in one or several messages being put into the kernel ring buffer for every received or transmitted packet leading to the growth of the log files at about 20 MiB/s. A solution to this could be to limit the size of the log files but it was simpler to shut-down the syslog service altogether (e.g., `systemctl stop systemd-journald && systemctl stop systemd-journald.socket && systemctl mask systemd-journald`). After this the performance was better but still lower than that of tests done by others. Therefore we chose to recompile the kernel modules without the debugging options.

4.3 Scenario 2: routing tests

To test the routing capabilities a more elaborate test network consisting out of four nodes was created. Even though it should be possible to use four different VLANs on a single virtual bridge to connect the VMs, influencing and troubleshooting VLANs from the outside was deemed to be more complex. We are mainly concerned in routing on within the physical layer (N-1) therefore only one normal DIF is used. A network consisting of four systems was chosen due to the ability to test the way redundant interfaces and paths are utilised in the IRATI implementation. This test setup also enables the testing of multipath routing. In Figures 4.3 and 4.4 visual representations of the setup are given.

4.3.1 Test cases

The following tests were conducted:

- Carrying out the same tests as during the first test scenario but with a stronger focus on the routing behaviour and data from the RIBs.
- Simulating the failure of an interface, either by shutting it down within the VM, shutting down the bridge or blocking traffic on the bridge. During this test `rina-echo-time` is used while the functioning of virtual network interface gets interrupted. If this results in a persistent failure of the program, the program will be restarted to check whether it is able to resume its correct behaviour.

4.3.2 Expected results

We expected the tests to have the following outcomes:

- Since this design seems to be a continuation of the second tutorial [46] we expected the setup to behave similarly during the start-up of the stack. However, because this network contains a cycle there is a slight possibility that this will cause technical issues (e.g., enrolling remote IPCPs).
- Due to the way routing is implemented in the IRATI prototype we expected that link failure would cause a fast reroute of the flow. However, due the the experimental nature of the code base several problems could arise (e.g., failure to make correct routing decisions or even performance degradation due to routing loops, although this should not happen due to the use of Loop Free Alternates Fast Reroute).

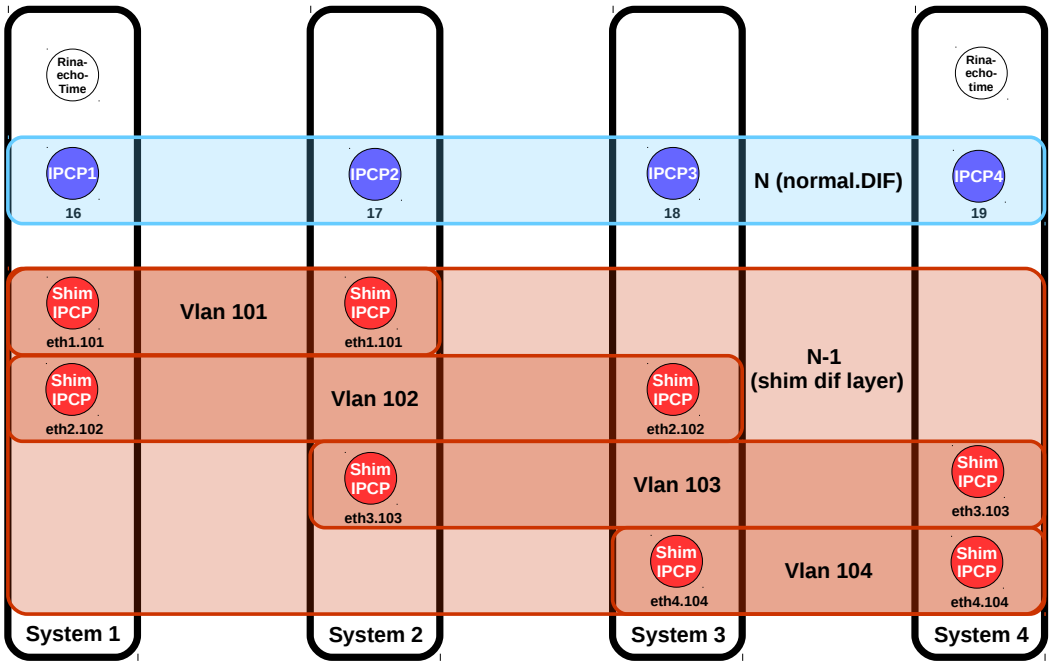

Figure 4.3: Logical design scenario 2

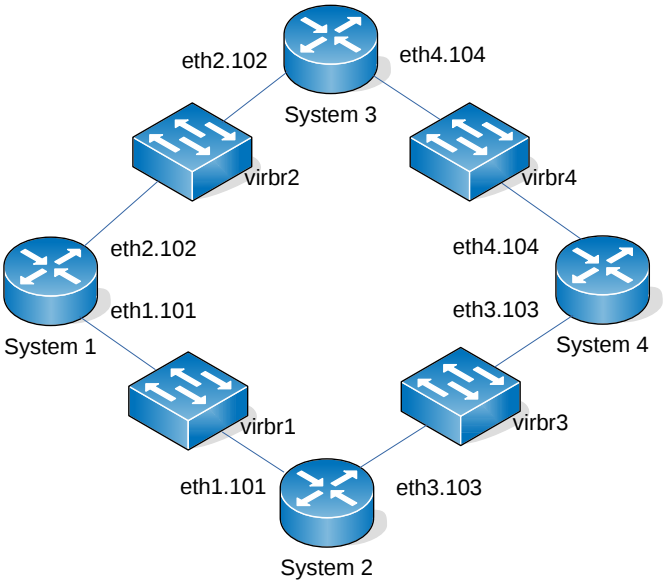

Figure 4.4: Physical design scenario 2

4.3.3 Encountered problems

- Setting up nodes with multiple shim DIFs involved some trial and error. During this endeavour we found out that IRATI currently does not support multiple IPCPs per DIF within a machine.[47] After studying the second and third tutorial we found that it was necessary to use a unique `apName` while creating the shim DIF IPCPs.
- Contrary to the expectations the flows were not relayed while physical paths were interrupted. We tried whether different `enrollmentTaskConfiguration` resulted in different outcomes, with timings like a `watchdogPeriodInMs` of `5000`, a `declaredDeadIntervalInMs` of `15000` and a `neighborsEnrollerPeriodInMs` of `30000` but this was not the case. We also tried whether the multipath plugin resulted in the expected outcome and while multiple links were successfully utilised simultaneously it did not improved the resilience.

  After investigating the cause for this behaviour we found that during the Pristine project the Loop Free Alternates feature of the IP Fast Reroute specification has been tested successfully. Therefore we tried to reproduce this experiment. This required changes to the test scenario which are shown in Section 4.3.4.

4.3.4 Alternative setup: Loop Free Alternates

According to the "Loop Free Alternates in RINA" experiment flows should be resilient to link failures while tolerating approximately 100 milliseconds of disruption. In the experiment only flows without flow control were successfully rerouted so the same configuration was used in the alternative setup.[30, p. 29] In Figures 4.5 and 4.6 visual representations of the setup are given.

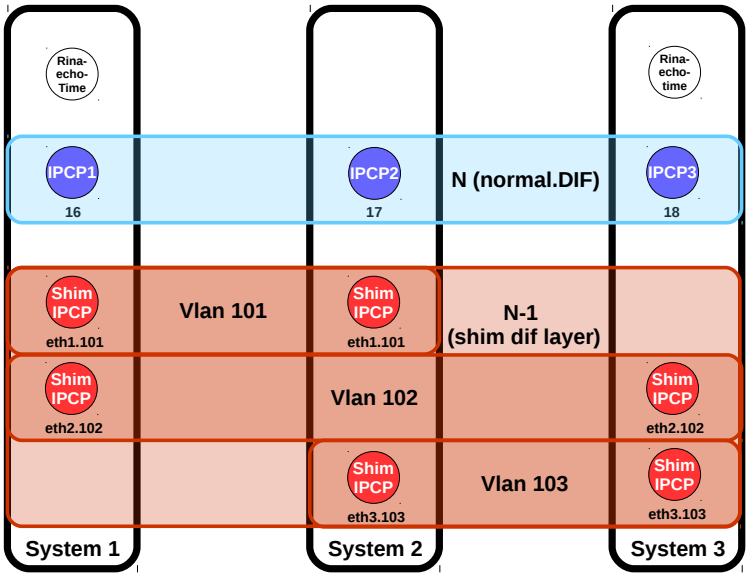

Figure 4.5: Logical design scenario 2

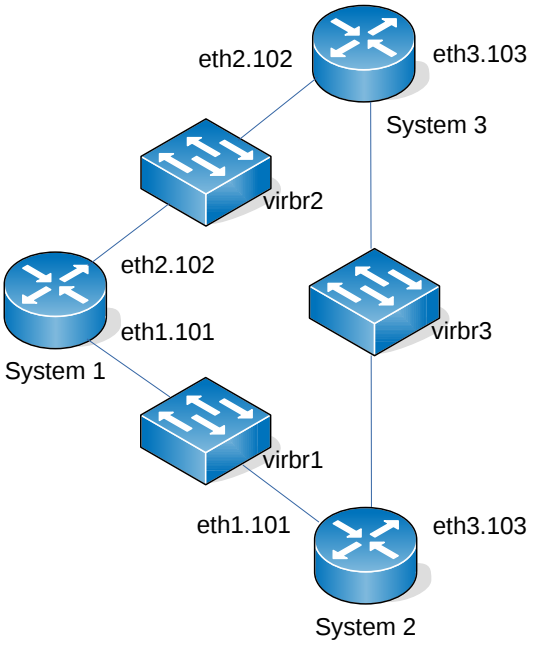

Figure 4.6: Physical design scenario 2

5 Results

In this chapter, the results of the experiments are given. The results originate from the two test scenarios that are defined in the experiments (see Chapter 4).

5.1 Scenario 1

Scenario 1 consists of a basic network with two virtual machines connected to each other (see Figures 4.1 and 4.2).

5.1.1 IRATI stack test

The IPC manager requires configuration from the files *ipcmanager.conf*, *default.dif* and *shim-eth-vlan.dif*. System 1 and System 2 have similiar configurations with the only difference being `test1.IRATI` being configured on System 1 and `Test2.IRATI` on System 2. The underlying shim DIFs are automatically registered to `test1.IRATI` and `Test2.IRATI` (see Listing 2). The IPCM log identicates this automatic registration request to the normal IPCP of the system. If the interfaces of systems are connected in the same VLAN they are considered to be enrolled automatically.[20, p. 57] .

```
DIF assignment operation completed for IPC process test2.IRATI:1::
↪  [success=1]

[register_at_dif]: Requested DIF registration of IPC process
↪  test2.IRATI:1:: at DIF 110::: through IPC process test-eth-vlan:1::

[ipcm_register_response_ipcp]: IPC process test2.IRATI:1:: informed about
↪  its registration to N-1 DIF 110:::

resource-allocator (INFO): IPC Process registered to N-1 DIF 110
```

Listing 2: Registration of IPCP to DIF 110

Next a normal DIF is created which will handle the communication between applications. Only a normal DIF can be used for communication applications. The normal DIF consists of a normal IPCP which is assigned to the DIF. When the IPCP list is consulted the overview of the current registrations and assignments of DIFs and IPCPs is given. The enrolment of `test1.IRATI` is done through the underlying shim DIF `110`. By querying the RIB it is possible to see that the underlying shim DIF is 110 (see Listing 4). The underlying shim DIF relays the enrolment request from IPCP `test1.IRATI` on System 1 to `test2.IRATI` on System 2. This is also visible in the log of both systems (Listing 7). On system 2 it shows a similar event happening with the difference that a flow allocation request has arrived from System 1.

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

Listing 3: Output of list-ipcps IPCM console command

```
Name: /difmanagement/enrollment/neighbors/processName=test2.IRATI; Class:
↪  Neighbor; Instance: 34
Value: Name: test2.IRATI-1--; Address: 17; Enrolled: 1
; Supporting DIF Name: 110; Underlying port-id: 2; Number of enroll.
↪  attempts: 0
```

Listing 4: Underlying DIF of the IPCP

5.1.2 Wireshark

The IRATI implementation includes a Wireshark fork with EFCP and CDAP dissectors. The capture of every experiment consist of malformed packets and supposedly non-malformed watchdog PDUs. Wireshark does not detect these watchdog PDUs as malformed, however the fields do contain inconsistent values. The watchdogs function as a timer and are sent periodically for Flow Liveness Detection (FLD) to detect if a flow between IPCPs is still alive [48, p. 107]. Every system that is a member of a DIF is sending watchdog PDUs to indicate that its alive. If the watchdogs do not arrive and the timer expires the neighbour is declared dead. The EFCP PDUs contains the following fields (see Listing 5).

```
PDU Type: Unknown (0x00000001)
Destination address: 18
Source address: 16
Destination Connection Endpoint ID: 1
Source Connection Endpoint ID: 0
Quality of Service ID: 0
PDU Flags: 64
Sequence number: 146176
ACK/NACK sequence number: 134217728
New right window edge: 403443712
New left window edge: 1627793920
Left window edge: 1952539743
Right window edge: 1627796065
Last control sequence received: 1952539743
```

Listing 5: Fields of efcp protocol

- PDU type: ACK/NACK/Flow, Selective ACK/NACK/Flow, or Control ACK. The first PDU type indicates the normal acknowledgement of a PDU/flow, the second PDU type is an optimized form and the control PDU type can be used for synchronization recovery.[49]
- Destination address: address of the destination.IPCP
- Source address: address of the source IPCP where the PDU originates.
- Destination Connection Endpoint ID: end point of a flow.[49]
- Source Connection Endpoint ID: begin point of a flow.[49]
- Quality of Service ID: identifier within a DIF that identifies a QoS-hypercube.[49]
- PDU Flags: conditions that affect the handling of the PDU.[49]
- Sequence number: signifies the number of the PDU.[49]
- ACK/NACK Sequence number: signifies the number of the PDU acknowledgement.[49]
- New right window edge: allows the right window edge to be updated.[49]
- New left Window edge: lower bound for all of the sequence numbers in the ACK/NACK list or confirming the previous the previous Left window edge.[49]
- Left window edge: last PDU ACK'ed or ACK-received.[1, p. 51]
- Right window edge: right end of the send window, the largest sequence number that the sender can send.[1, p. 51]
- Last control sequence received: sequence number of the last PDU with PDU type control.[1]

A flow allocation request is visible in the Wireshark capture. The capture shows two PDUs that indicate a flow allocation request is sent between two systems (see Listing 6). The application that is registered is the `rina-echo-time` server and client.

```
Error and Flow Control Protocol, Unknown (0x1201) PDU
PDU Type: Unknown (0x00000001)
Destination address: 18
Source address: 16
Destination Connection Endpoint ID: 1
Source Connection Endpoint ID: 0
Quality of Service ID: 0
PDU Flags: 64
Sequence number: 146176
ACK/NACK sequence number: 134217728
New right window edge: 403443712
New left window edge: 1627793920
Left window edge: 1952539743
Right window edge: 1627796065
Last control sequence received: 1952539743

0000  52 54 00 38 72 65 52 54 00 68 57 26 81 00 00 66   RT.8reRT.hW&...f
0010  d1 f0 01 12 00 10 00 01 00 00 00 00 00 40 00 3b   .............@.;
0020  02 00 00 00 00 08 00 10 0c 18 00 2a 06 61 5f 64   ...........*.a_d
0030  61 74 61 32 06 61 5f 64 61 74 61 38 00 42 e6 03   ata2.a_data8.B..
0040  32 e3 03 08 10 10 12 1a dc 03 08 00 10 05 18 01   2...............
0050  2a 04 46 6c 6f 77 32 12 2f 66 61 2f 66 6c 6f 77   *.Flow2./fa/flow
0060  73 2f 6b 65 79 3d 31 38 2d 33 38 00 42 90 03 32   s/key=18-38.B..2
0070  8d 03 0a 22 0a 19 72 69 6e 61 2e 61 70 70 73 2e   ..."..rina.apps.
0080  65 63 68 6f 74 69 6d 65 2e 63 6c 69 65 6e 74 12   echotime.client.
0090  01 31 1a 00 22 00 12 22 0a 19 72 69 6e 61 2e 61   .1.."..".."rina.a
00a0  70 70 73 2e 65 63 68 6f 74 69 6d 65 2e 73 65 72   pps.echotime.ser
00b0  76 65 72 12 01 31 1a 00 22 00 18 03 20 03 28 12   ver..1..".... .(.
```

Listing 6: Flow allocation request

Initially the Wireshark implementation had the wrong field content (see Figure 5.1). The Wireshark dissector is configured to have 4 byte addresses by default but the addresses in the experiments use addresses of 2 bytes. To resolve this issue the user can adjust the addresses in the configuration or adjust the dissector. Wireshark is unable to detect the settings of the IRATI stack and adapt accordingly, thus Wireshark is unable to correctly interpret the bytes. We decided to create a patch which adjusts the byte offset (see Figure 5.1) in the dissector per field to a similar sizes as in the IRATI configuration files. The patch however is unable to solve every issue in the dissector and not every field is verifiable. The source and destination address are however according to the pre-defined settings.

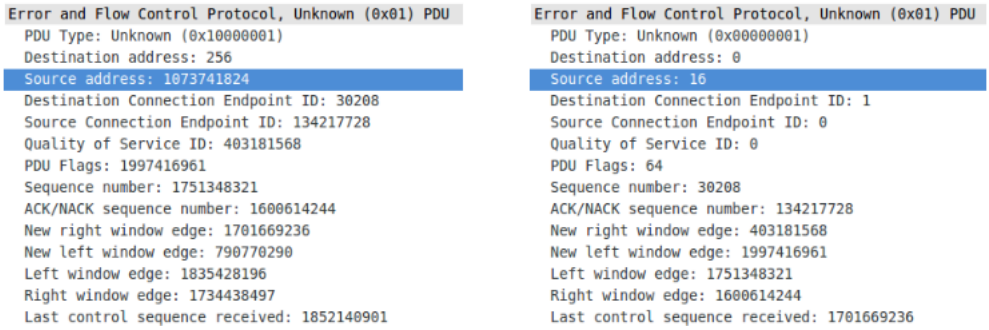

Figure 5.1: Wireshark without patch (left side) and with patch (right side)

```
[enroll_to_dif]: Requested enrollment of IPC process test1.IRATI:1:: to
↪  DIF normal.DIF::: through DIF 101::: and neighbor IPC process
↪  test2.IRATI:1::

librina.irm (INFO): Requested the allocation of N-1 flow to application
↪  test2.IRATI-1 through DIF 101
ipcm.flow-alloc (INFO)[flow_allocation_requested_local]: IPC process
↪  test-eth-vlan1:1:: requested to allocate flow between test1.IRATI:1::
↪  and test2.IRATI:1::


ipcm.flow-alloc (INFO)[ipcm_allocate_flow_request_result_handler]:
↪  Informing IPC process test-eth-vlan1:1:: about flow allocation from
↪  application test1.IRATI:1:: to application test2.IRATI:1:: in DIF
↪  101::: [success = 1, port-id = 1]

internal-events (INFO): Event N_MINUS_1_FLOW_ALLOCATED has just happened.
↪  Notifying event listeners.
ipcm.flow-alloc (INFO)[ipcm_allocate_flow_request_result_handler]:
↪  Applications test1.IRATI:1:: and test2.IRATI:1:: informed about flow
↪  allocation result

rib (INFO): Bound port_id: 1 CDAP connection to RIB version 1 (AE
↪  Management)
internal-events (INFO): Event NEIGHBOR_ADDED has just happened. Notifying
↪  event listeners.
ipcp[3].routing-ps-link-state (INFO): There was an allocation flow event
↪  waiting for enrollment, launching it
ipcm.ipcp (INFO)[enroll_to_dif_response_event_handler]: Enrollment
↪  operation completed for IPC process test1.IRATI:1::

959(1453721733)#ipcp[3].enrollment-task-ps-default (INFO): Remote IPC
↪  Process enrolled!
```

Listing 7: Enrolment request

5.1.3 Connectivity test

The starting point of this test case is:

- IRATI stack is started
- Enrolment Phase of a normal DIF has been executed

The connectivity test consists of using the `rina-echo-time` tool, which is a client-server application and has similarities to the Internet Control Message Protocol (ICMP) ping tool. The first step is to start the server application and then the client. During the experiments System 1 was running the server that listens for SDUs, and System 2 is using the client which sends SDUs. When starting the server on System 1 the log of the IPCM application shows that the `rina-echo-time` tool is requesting a registration to the IPCP of the normal DIF (see Listing 8).

```
ipcm.app (INFO)[app_reg_req_handler]: Requested registration of
↪  application rina.apps.echotime.server:1:: to IPC process
↪  test1.IRATI:1::

ipcp[3].namespace-manager (INFO): Successfully registered application
↪  rina.apps.echotime.server-1-- with IPC Process id 3s
ipcm.app (INFO)[notify_app_reg]: Application rina.apps.echotime.server:1::
↪  informed about its registration to N-1 DIF normal.DIF::: [success = 1]
```

Listing 8: Registration of the rina-echo-time tool is successful

When the client on System 2 is launched a flow allocation process is started. It immediately results in a success and a port-id is assigned to the flow to identify the instance of the `rina-echo-time` tool in the normal DIF (see Listing 9).

```
ipcm.flow-alloc (INFO)[flow_allocation_requested_local]: IPC process
↪  Test2.IRATI:1:: requested to allocate flow between
↪  rina.apps.echotime.client:1:: and rina.apps.echotime.server:1::

ipcm.flow-alloc (INFO)[ipcm_allocate_flow_request_result_handler]:
↪  Informing IPC process Test2.IRATI:1:: about flow allocation from
↪  application rina.apps.echotime.client:1:: to application
↪  rina.apps.echotime.server:1:: in DIF normal.DIF::: [success = 1,
↪  port-id = 4]

ipcm.flow-alloc (INFO)[ipcm_allocate_flow_request_result_handler]:
↪  Applications rina.apps.echotime.client:1:: and
↪  rina.apps.echotime.server:1:: informed about flow allocation result
```

Listing 9: Client flow allocation on System 2

Finally when the IPCP list is queried on both System 1 and System 2, the flow seems to be visible on System 1 and the registered AP as well (see Listing 10). However, on System 2 the client application is not visible (see Listing 11). The IPCP lists of both systems show the port-id of the registered application `rina-echo-time`. The port-id of the tool on System 2 is only visible when the tool is sending PDUs.

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
↪  | Port-ids of flows provided)
   1 | test-eth-vlan:1:: | shim-eth-vlan | ASSIGNED TO DIF 110 |
↪  test1.IRATI-1-- | 2
   2 | test1.IRATI:1:: | normal-ipc | ASSIGNED TO DIF normal.DIF |
↪  rina.apps.echotime.server-1-- | 3
```

Listing 10: IPCP list of System 1 that shows the server

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications
↪  | Port-ids of flows provided)
   1 | test-eth-vlan:1:: | shim-eth-vlan | ASSIGNED TO DIF 110 |
↪  test2.IRATI-1-- | 1
   2 | test2.IRATI:1:: | normal-ipc | ASSIGNED TO DIF normal.DIF | - | 2
```

Listing 11: IPCP list of System 2

When querying the RIB on both systems all source and destination addresses are visible. The actual performance of the `rina-echo-time` tool shows the round-trip time (RTT) of the SDUs. During the experimentation the tool typically indicates a RTT of lower than 1 ms (see Listing 12). The RTT does not seem completely consistent and begins with a value of 7.9788 ms (however, this behaviour of having a high initial RTT was consistently seen during the tests and might be attributable to the way flows are established). The capture of Wireshark identifies the PDUs, the payload indicates that the application `echo-time-tool` has requested a flow over the network. The PDUs that are sent however do not appear to be of substantial value.

```
root@System2:~# rina-echo-time -c 10
16320(1453131098)#librina.logs (DBG): New log level: INFO
16320(1453131098)#librina.nl-manager (INFO): Netlink socket connected to
 ↪  local port 16320
Flow allocation time = 4.0223 ms
SDU size = 20, seq = 0, RTT = 7.9788 ms
SDU size = 20, seq = 1, RTT = 0.50487 ms
SDU size = 20, seq = 2, RTT = 0.51597 ms
SDU size = 20, seq = 3, RTT = 0.47762 ms
SDU size = 20, seq = 4, RTT = 0.52016 ms
SDU size = 20, seq = 5, RTT = 0.51088 ms
SDU size = 20, seq = 6, RTT = 0.44791 ms
SDU size = 20, seq = 7, RTT = 0.51606 ms
SDU size = 20, seq = 8, RTT = 0.52716 ms
SDU size = 20, seq = 9, RTT = 3.9779 ms
SDUs sent: 10; SDUs received: 10; 0\% SDU loss
Minimum RTT: 0.44791 ms; Maximum RTT: 7.9788 ms; Average RTT:1.5977 ms;
 ↪  Standard deviation: 2.4941 ms
SDUs sent: 0; SDUs received: 0; 0\% SDU loss
Minimum RTT: 9.2234e+18 ms; Maximum RTT: 0 ms; Average RTT:0 ms; Standard
 ↪  deviation: -0 ms
```

Listing 12: Echo-time results

5.1.4 Performance test

The starting point of this test case is:

- IRATI stack is started
- Enrolment phase of a normal DIF has been executed
- Connectivity test has shown that there is a connection between System 1 and 2

Performance tests were done mainly to verify whether our setup was functioning properly and to check our results with those of earlier experiments. The performance tests were executed with two different virtual interfaces. These are the VirtIO virtual interface and the E1000 virtual NIC. The first tests were done with VirtIO and consisted of simply sending SDUs with a size of 500 bytes resulted in an average speed of around 40 Mbit/sec. IRATI has debugging options enabled by default which generate syslog files for at least every packet. This has an impact on the CPU of the machines and the goodput[1] was low. After turning off the debug options the goodput was increased to 200 Mbit/sec. The `rina-tgen` tool however indicated that some PDU's were dropped. Wireshark was running in the background which might affected the results. After Wireshark was disabled the goodput was 410 Mbit/sec.

The `rina-tgen` tool allows one to set the SDU size which is similar to setting another maximum transmission unit (MTU) and maximum segment size (MSS) size. To test whether

[1] We conform to the IRATI project and refer to the application level throughput as goodput.

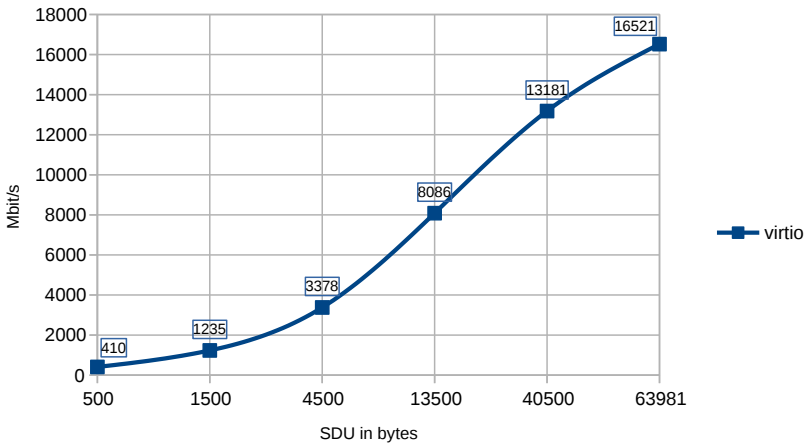

Figure 5.2: Relation between goodput and SDU size

this resulted in higher goodput the MTU size was set to 64k and the SDU size was multiplied by a factor of three for every new test run, starting from the default of 500 bytes (a factor of three was chosen solely to limit the amount the tests necessary), while each run consists of 10 measurements. The performance was tested with SDU sizes of 1,500, 4,500, 13,500, 40,500 and 63,981 bytes. As can be seen in the Table 5.1 the goodput increased from 410 Mbit/sec to 16520 Mbit/sec (see Figure 5.2). The same test environment with TCP/IP and Iperf results in the average goodput of 33 Gbit/sec (Iperf3 gave similar results).

The tests on the E1000 NIC resulted in a lower goodput than the VirtIO NIC tests (see Figure 5.3). The E1000 results are acquired from tests with the SDU sizes 500, 1,500, 4,500, 13,500 and 16,091. Just like with the VirtIO tests the SDU size is gradually increased. However, it is not possible to increase the MTU size past 16,110 bytes and when the SDU size is more than 16,091 bytes PDUs were dropped. Similar behaviour was seen with VirtIO, however the packets started to get dropped when the SDU size was higher than 63,981 (see Listing 13). These thresholds hint that the packets have an overhead of 19 bytes. The goodput of the E1000 went from 94 Mbit/sec to 2,434 Mbit/sec with the highest SDU size. When using a same test environment with TCP/IP and Iperf the average goodput with the E1000 virtual NIC was 3.8 Gbit/sec.

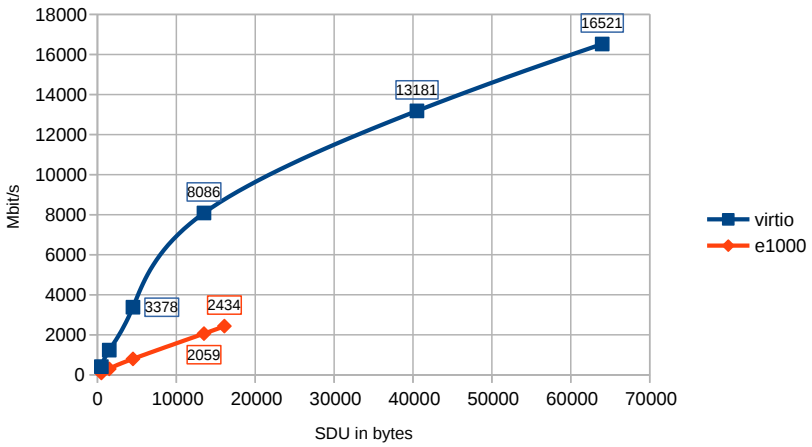

Figure 5.3: Difference in rina-tgen goodput between VirtIO and E1000

```
[27351.286809] rina-shim-eth-vlan(ERR): SDU too large (64001), dropping
[27351.286849] rina-rmt(ERR): Failed to send a PDU to port-id 1
[27351.286897] rina-shim-eth-vlan(ERR): SDU too large (64001), dropping
[27351.286937] rina-rmt(ERR): Failed to send a PDU to port-id 1
[27351.286985] rina-shim-eth-vlan(ERR): SDU too large (64001), dropping
[27351.287027] rina-rmt(ERR): Failed to send a PDU to port-id 1
```

Listing 13: Error message of the IPCM when the SDU size (63,982) is too large

SDU size	**amount of SDUs**	**Goodput**	**Speed**
500	1024806 SDUs	512403000 bytes in 10001202 us	409.8731 Mbit/s
1500	1029175 SDUs	1543762500 bytes in 9999122 us,	1235.1184 Mbit/s
4500	938262 SDUs	4222179000 bytes in 9999005 us	3378.0793 Mbit/s
13500	748599 SDUs	10106086500 bytes in 9999060 us	8085.6292 Mbit/s
40500	406792 SDUs	16475076000 bytes in 9999002 us	13181.3763 Mbit/s
63981	322733 SDUs	20648780073 bytes in 9999013 us	16520.6546 Mbit/s

Table 5.1: Performance test results

5.2 Scenario 2

These results are from a four node network with redundant paths. The network is set up according to Figures 4.3 and 4.4. Whereas the results of scenario 1 showed whether the RINA stack worked, the results of scenario 2 indicate the potential of routing in a network with redundant paths.

5.2.1 Connectivity test

Just like in scenario 1 the connectivity tests started with the registration of the underlying DIFs to the IPCPs. The difference is that the IPCP is registered to multiple underlying DIFs (see Listing 14). On System 1, System 2, System 3 and System 4 all registration were successful if the configuration had no mistakes. The enrolment of the IPCPs to the DIFs happened according to Figure 4.3.

The most visible differences in this scenario take routing into consideration. The RIB shows it is possible to see the next hops to reach the destination system (see Listing 15). The Listing 15 indicates the next hops from System 4 (with address 19), to reach System 1 (with address 16), traffic first has to go through System 2 (with address 17). Any stream of packets goes through the specific underlying shim DIF according to the next hop to reach the destination. In this case to reach address 16 from address 19, PDUs will be sent through DIF 103.

```
Name: /ipcmanagement/irm/underregs/difName=103; Class:
↪ UnderlayingRegistration; Instance: 31
Value: N-1 DIF name: 103

Name: /ipcmanagement/irm/underregs/difName=104; Class:
↪ UnderlayingRegistration; Instance: 32
Value: N-1 DIF name: 104
```

Listing 14: Output of the RIB showing the underlying DIFs of System 4

```
Name: /resalloc/nhopt/key=16-0; Class: NextHopTableEntry; Instance: 47
Value: Destination address: 16; QoS-id: 0; Cost: 1; Next hop addresses:
↪ 17/

Name: /resalloc/nhopt/key=17-0; Class: NextHopTableEntry; Instance: 48
Value: Destination address: 17; QoS-id: 0; Cost: 1; Next hop addresses:
↪ 17/

Name: /resalloc/nhopt/key=18-0; Class: NextHopTableEntry; Instance: 49
Value: Destination address: 18; QoS-id: 0; Cost: 1; Next hop addresses:
↪ 18/
```

Listing 15: Next hops to several destinations from System 4

Besides the next hops it is possible to see the characteristics of all systems in the network (i.e. a DIF). So-called Flow State Objects are exchanged between IPCPs in order to disseminate the state of the supporting N-1 flows (see Listing 16).[48, p. 117]

```
Name: /resalloc/fsos/key=16-17; Class: FlowStateObject; Instance: 37
Value: Address: 16; Neighbor address: 17; cost: 1
Up: 1; Sequence number: 1; Age: 2887

Name: /resalloc/fsos/key=16-18; Class: FlowStateObject; Instance: 38
Value: Address: 16; Neighbor address: 18; cost: 1
Up: 1; Sequence number: 1; Age: 2887

Name: /resalloc/fsos/key=17-16; Class: FlowStateObject; Instance: 39
Value: Address: 17; Neighbor address: 16; cost: 1
Up: 1; Sequence number: 1; Age: 2887

Name: /resalloc/fsos/key=17-19; Class: FlowStateObject; Instance: 40
Value: Address: 17; Neighbor address: 19; cost: 1
Up: 1; Sequence number: 1; Age: 167

Name: /resalloc/fsos/key=18-16; Class: FlowStateObject; Instance: 41
Value: Address: 18; Neighbor address: 16; cost: 1
Up: 1; Sequence number: 1; Age: 2887

Name: /resalloc/fsos/key=18-19; Class: FlowStateObject; Instance: 46
Value: Address: 18; Neighbor address: 19; cost: 1
Up: 1; Sequence number: 1; Age: 167

Name: /resalloc/fsos/key=19-17; Class: FlowStateObject; Instance: 36
Value: Address: 19; Neighbor address: 17; cost: 1
Up: 1; Sequence number: 1; Age: 167

Name: /resalloc/fsos/key=19-18; Class: FlowStateObject; Instance: 45
Value: Address: 19; Neighbor address: 18; cost: 1
Up: 1; Sequence number: 1; Age: 167
```

Listing 16: Flow State Objects of all IPCPs

5.2.2 Performance test

The performance tests from System 1 to System 4 show a decrease in goodput compared to the first test scenario. Where in scenario 1 a transmission of SDUs with the size of 500 bytes resulted in about 410 Mbit/sec, in scenario 2 the result did not reach 280 Mbit/sec. The highest speed with an SDU size of 63,981 bytes is 7,756 Mbit/sec. Table 5.2 shows all results regarding the performance test of scenario 2.

SDU size	amount of SDUs	Goodput	Speed
500	691507 SDUs	345753500 bytes in 9999522 us	276.6160 Mbit/s
1500	662775 SDUs	662775 bytes in 9999085 us,	795.4028 Mbit/s
4500	573035 SDUs	2578657500 bytes in 9999236 us	2063.0836 Mbit/s
13500	438390 SDUs	5918265000 bytes in 9999624 us	4734.7900 Mbit/s
40500	208678 SDUs	8451459000 bytes in 9999439 us	6761.5465 Mbit/s
63981	150851 SDUs	9651597831 bytes in 9999458 us	7721.6968 Mbit/s

Table 5.2: Performance test results

5.2.3 Link failure test

The earlier results show that the RIB has knowledge about all connected systems. This would create the assumption that a when a link fails traffic is able to be rerouted through another next known hop. First experiments with link failure consists of disconnecting a link between System 1 and System 3 while `rina-echo-time` is running on System 1 and 4. The ongoing flow continued without any issues after the disconnection. Analysis showed however that the flow was going through System 2 and disconnecting the link between System 1 and 3 has no impact if the link is not used.

When disconnecting only the link between System 1 and System 2 the flow was discontinued. When querying the RIB of System 1 after a minute, to allow the propagation of network updates, it showed changes to the next hops (see Listing 19). The next hop to System 4 changed from System 2 to System 3. System 2 is completely removed from the next hop list in System 1's RIB (see Listing 19). The RIB changes indicate that updates in the network are propagated to every system. After the link between System 1 and 2 got disconnected these systems were unable to communicate with each other. System 1 is also unable to communicate with System 4 even though the RIB indicates a next hop.

The last test consisted of starting a new flow with the `rina-echo-time` tool. Important in this test is the recovery of the network. An ongoing flow might not reroute through a different path but a new flow might take a different path. System 1 was unable to allocate a flow request to 4, thus no communication was possible. System 1 was also unable to send PDUs to System 2. The only system that is reachable for System 1 is the directly connected System 3.

When analysing the log on System 2 the event *neighbour_declared_dead* is visible (see Listing 17). This event follows with a deallocation of the flow through shim IPCP test-eth-vlan:1 which is connected to System 1.

```
ipcp[3].enrollment-task (ERR): Problems generating or sending CDAP
↪  message: The invokeid 1 already exists
ipcp[3].enrollment-task (ERR): Problems generating or sending CDAP
↪  message: The invokeid 1 already exists
internal-events (INFO): Event NEIGHBOR_DECLARED_DEAD has just happened.
↪  Notifying event listeners.
ipcp[3].enrollment-task (INFO): Requesting the deallocation of the N-1
↪  flow with the dead neighbor
ipcm.flow-alloc (INFO)[deallocate_flow]: Application IPCP2.IRATI:1:: asks
↪  IPC process test-eth-vlan:1:: to deallocate flow [port-id = 1]
```

Listing 17: Neighbour declared dead on System 2

```
Name: /resalloc/nhopt/key=17-0; Class: NextHopTableEntry; Instance: 100
Value: Destination address: 17; QoS-id: 0; Cost: 1; Next hop addresses:
↪  17/

Name: /resalloc/nhopt/key=18-0; Class: NextHopTableEntry; Instance: 101
Value: Destination address: 18; QoS-id: 0; Cost: 1; Next hop addresses:
↪  18/

Name: /resalloc/nhopt/key=19-0; Class: NextHopTableEntry; Instance: 102
Value: Destination address: 19; QoS-id: 0; Cost: 1; Next hop addresses:
↪  17/
```

Listing 18: Next hops of System 1 before failure

```
Name: /resalloc/nhopt/key=18-0; Class: NextHopTableEntry; Instance: 155
Value: Destination address: 18; QoS-id: 0; Cost: 1; Next hop addresses:
↪  18/

Name: /resalloc/nhopt/key=19-0; Class: NextHopTableEntry; Instance: 156
Value: Destination address: 19; QoS-id: 0; Cost: 1; Next hop addresses:
↪  18/
```

Listing 19: Next hops of System 1 after failure

5.2.4 Multipath plugin

The multipath plugin was tested to see whether it would introduce a workaround for the routing resilience. The plugin required manual compilation. After installing the plugin and configuring the test environment equivalent to scenario 2 the RIB of System 1 indicates changes to the next hops between System 1 and System 4. To reach System 4 (with address 19) the next hops are 17 and 18 which are systems 2 and 3. (see Listing 20).

```
Name: /resalloc/nhopt/key=17-1; Class: NextHopTableEntry; Instance: 51
Value: Destination address: 17; QoS-id: 1; Cost: 1; Next hop addresses:
↪  17/

Name: /resalloc/nhopt/key=18-1; Class: NextHopTableEntry; Instance: 53
Value: Destination address: 18; QoS-id: 1; Cost: 1; Next hop addresses:
↪  18/

Name: /resalloc/nhopt/key=19-1; Class: NextHopTableEntry; Instance: 52
Value: Destination address: 19; QoS-id: 1; Cost: 2; Next hop addresses:
↪  17/ 18
```

Listing 20: Multipath plugin next hops

Both the tools `rina-echo-time` and `rina-tgen` send the PDUs from its source to the destination through all available paths. By using Wireshark it is possible to see the utilisation of the multiple paths by inspecting the VLAN-IDs.

Finally the resilience test is done by disconnecting specific links. The link between System 1 and System 2 is disconnected and the flow seems to be stopped. `rina-echo-time` indicates a timeout (see Listing 21) and repeats this timeout message for the remaining amount of sent PDUs.

The earlier results show that the RIB has knowledge about all connected systems. This would create the assumption that a when a link fails traffic is able to be rerouted through another next known hop. First experiments with link failure consists of disconnecting a link between System 1 and System 3 while `rina-echo-time` is running on System 1 and 4. The ongoing flow continued without any issues after the disconnection. Analysis showed however that the flow was going through System 2 and disconnecting the link between System 1 and 3 has no impact if the link is not used.

```
root@rina4:~# rina-echo-time -c 10
19973(1453301234)#librina.logs (DBG): New log level: INFO
19973(1453301234)#librina.nl-manager (INFO): Netlink socket connected to
↪  local port 19973
Flow allocation time = 4.721 ms
                SDU size = 20, seq = 0, RTT = 0.68067 ms
                SDU size = 20, seq = 1, RTT = 0.4919 ms
                SDU size = 20, seq = 2, RTT = 0.73065 ms
                SDU size = 20, seq = 3, RTT = 1.0135 ms
                SDU size = 20, seq = 4, RTT = 0.66604 ms
                SDU size = 20, seq = 5, RTT = 0.537 ms
19973(1453301242)#rina-echo-time (WARN): Timeout waiting for reply, SDU
↪  considered lost
19973(1453301244)#rina-echo-time (WARN): Timeout waiting for reply, SDU
↪  considered lost
19973(1453301246)#rina-echo-time (WARN): Timeout waiting for reply, SDU
↪  considered lost
19973(1453301248)#rina-echo-time (WARN): Timeout waiting for reply, SDU
↪  considered lost
SDUs sent: 10; SDUs received: 6; 40% SDU loss
Minimum RTT: 0.4919 ms; Maximum RTT: 1.0135 ms; Average RTT:0.68663 ms;
↪  Standard deviation: 0.18416 ms
SDUs sent: 0; SDUs received: 0; 0% SDU loss
Minimum RTT: 9.2234e+18 ms; Maximum RTT: 0 ms; Average RTT:0 ms; Standard
↪  deviation: -0 ms
```

Listing 21: Timeout after disconnecting link

5.2.5 Alternative setup results

The alternative setup consisted of recreating one of the experiment that was conducted during the PRISTINE project. In the experiment the flows without flow control were successfully rerouted. In the experiment conducted during the PRISTINE project flow control had to be disabled because the EFCP component seemed to stop working when too many PDUs are lost during. Therefore, we configured our setup according to the configuration of aforementioned experiment. The further tests proceeded equivalent to the previous link failure scenario. The results were also equivalent; the ongoing flow and new flow did not reroute through a different path.

6 Discussion and recommendations

In this chapter the results from Chapter 5 are discussed and recommendations are given.

6.1 Connectivity

The results of the connectivity tests were positive. All the server applications were able to register themselves in the DIF underneath them and the clients on other systems were able to connect to the listening servers. It was possible to manually change the name of the application used to register with and the clients could still connect. Besides supporting one client per server application it was possible to create multiple client flows to the same server.

Prior to launching the IPCM it was required to properly set up the interfaces, but it was not necessary to configure the links of all other systems. Only during enrolment the other systems need to be in the right state (i.e. configured interfaces and a running IRATI stack).

Wireshark showed that after the `enroll-to-dif` command is issued, packets are exchanged between the two systems. After that watchdog packets are regularly exchanged, according to the timers set, which are used to determine whether the flow is still alive (and in the case of a flow between IPCPs whether the neighbour is still up). When IPCPs were not enrolled to a remote DIF there were no packets exchanged between these systems. Updates to the RIB were propagated after a new flow was requested and before a new flow was established.

6.2 Performance

The results of the performance test, which were held mainly to make sure the test environment was set-up correctly, were positive. In the virtual environment the stack was able to achieve a top mean bandwidth of 16.5 Gbit/sec with an MTU size of 64 kilobytes and an SDU size of 63,981 bytes. Compared to the 33.5 Gbit/sec of TCP/IP (tested via Iperf) this may seem a bit low but it needs to be considered that the IRATI stack has not been optimised yet for speed (even though performance was one of its design considerations) and it is unlikely that the optimisations of the virtualisation stack for TCP/IP also benefit RINA to the same extent. These results make us believe that the IRATI implementation is on the right track in terms of performance.

Through our tests it is hard to determine the bottleneck of the stack and to make predictions about where the limits lie. During the achievement of the highest goodput the `rina-echo-time` application seemed to max out one CPU core. This could indicate that the test application in user space might be a bottleneck. The difference between the VirtIO and E1000 NICs can be attributed to the advantages the paravirtualised VirtIO NIC has over normal emulated hardware.

6.3 Routing

The results of the routing resiliency tests were negative. Both the `rina-tgen` and `rina-echo-time` tools did not continue without interrupts when the "next hop" link was brought down. When putting down an unused redundant link it did not affect the network in any way. When a utilised link was put down the flow halted. Only when quickly re-enabling the link did the flow continue. While the link was down it was not possible to establish new connections between the nodes that shared the link. When the other node was considered down after the `declaredDeadIntervalInMs` timer expired the IPCP needed to be re-enrolled which appears to be expected behaviour.

It was tested whether the multipath plugin resulted in different behaviour and while it did successfully utilise multiple NICs simultaneously, it did not improve the resiliency; when one of the utilised links was brought down the flow halted altogether, even on the unaffected links.

After contacting Dimitri Staessens, he confirmed that an already created flow does not automatically get rerouted. He however mentioned that creating a new flow, thus starting a new instance of `rina-echo-time`, should be able to succeed. There is a report[30, p. 29] that confirms Mr Staessens statements. In the report experiments are conducted that validate the working of the Loop Free Alternates protocol which shows that an active flow is rerouted through an alternative path after a failure of a node or a link. While this might have worked in the experiments done during the Pristine project, this however did not occur in our experiments. Even after recreating the network of the report with the same configuration, the flows did not resume correctly over a different path.

In our experiments, traffic was not rerouted when a link failed and we have not been able to identify the source of this problem. There is a possibility that the inability to reroute is caused by a configuration problem on our side. However, from the other tests there is no indication of such a configuration problem. Further research on this subject is necessary to exactly determine the cause. This issue will be raised at the developers of IRATI which will hopefully lead to a solution either in the form of a configuration change on our side or in a software patch.

6.4 Generic topics

During the experiments it was difficult to get a clear view into the traffic flows due to the Wireshark dissectors being unable to dynamically adjust to the IRATI configurations. However, we were able to get a grasp of the intended behaviour because many packets included strings that indicated which operations they were supposed to do. The dissectors should be fairly easy to fix, as showed by the premature patch made by us, but due to the configurable lengths of the fields this is not a definite solution.

Furthermore we found it difficult to get insight in the activities within the DIFs and IPCPs. When the debugging options were enabled we encountered information overload but when it was disabled we had to work with the sparse and on first sight cryptic logging output of the IPCMs. We believe that the formatting of the RIB is adequate for programmers at this point of the development process but reckon that networking engineers without a programming background would favour a more routing table-like structure.

We found that the documentation on IRATI and RINA as a whole was quite dispersed which made it difficult to approach the topic. This can be attributed to the different research

projects that have contributed to the stack. In case of RINA this can be attributed to the way the reference model is developed behind closed doors. While references to a "RINA specification handbook" are made in the IRATI documents this handbook does not seem to be available. After asking the IRATI developers it was stated that this specification handbooks is actually the reference model (which consists of several documents) and from which an older version appears to be available in the RINA Specs Wiki.[28]

6.5 Recommendations

We make the following recommendations to the developers of IRATI and to researchers that are planning to build an experimental RINA/IRATI test environment:

- The first tests with the `rina-tgen` tool resulted in the flooding of the kernel ring buffer which caused low performance and the filling up of the disk. We found this is because the kernel modules are compiled with debugging options enabled by default. Even though the software is experimental we advise against this choice and would recommend to make debugging a runtime configurable option (e.g., via kernel module parameters). Mentioning this on the IRATI wiki could also help to a great extent.
- Setting up the IRATI stack through JSON configuration files and controlling the runtime via the IPCM console is quite cumbersome. It is trivial to make mistakes in the configuration files which often results in cryptic error messages or even kernel crashes. The IPCM is accessible via a Telnet console which makes it fairly hard to carry out repetitive task and does not provide useful feedback in case of incorrect usage. We worked around the former problem by wrapping certain console commands in bash and netcat constructs so that it was easier to use (see Appendix C.1).
- The documentation on IRATI and on RINA in general are quite dispersed. This makes it unnecessarily hard to quickly grasp the workings of RINA and determine its detailed characteristics. One explanation for this is that the topic is still fairly young and that the research initiatives have been carried out over several separate projects. Furthermore the reference model does not appear to be publicly accessible. We advise the Pouzin Society to make an inventory of all the documents produced by the projects it oversees and clearly list these sorted chronologically per project. We advise the developers of IRATI to start working on an in-tree manual (that includes descriptions of the tools) and to make sure the tutorials are up to date.
- For researchers that are not entirely sure which implementation to use we make the following advices: to quickly and easily learn about RINA, RINASim seems to be the right implementation to start with. Advantages are that it is reasonably stable and comes in a fully configured virtual machine. Disadvantages could be that it is not possible to use it with real programs and that it might disclose more information than that which is actually used or relevant to a real RINA implementation (e.g., viewing the actual content of packets might not be straight forward). For experimenting with a real implementation, that also supports actual programs, ProtoRINA seems to be a good choice. Advantages are that it is written in Java, thus it should be highly unlikely to cause kernel crashes. Disadvantages are that the performance is less likely to match that of an in-kernel implementation and the development takes place behind closed doors. Therefore, when performance and open development are important, IRATI appears to be the better choice.

7 Conclusion and future work

In this chapter we state our conclusions and finish the report with a selection of possible future work topics.

7.1 Conclusion

From our research we conclude that the IRATI implementation is still in an experimental state. Judged by the performance results, which approached half that of TCP/IP, the architecture of the IRATI implementation appears to be in the right direction. In terms of routing IRATI accommodates in the basic requirements of the RINA model but many components proposed to enable and enhance scalability are not yet in place. We verified that the multipath plugin correctly enables equal cost multipath routing but in terms of routing resiliency the implementation was not functioning as expected.

Even when taking the occasional crashes into account, that mainly happened when configuration errors were made, we deem the implementation ready to be studied by external researchers, because in general the functionality appears to stable enough. However, because these crashes in some cases also affect the kernel and because the configuration of the implementation is not straight forward, we believe that knowledge of Unix is required and that a background in programming is preferable. Due to aforementioned issues we do not think the implementation is ready for use outside the academic field at this moment in time.

7.2 Future work

We consider the following topics suitable for further research:

- We have built two test networks. These networks were of a rather simplistic nature since they only consisted of two layers of DIFs. Research into the behaviour and scalability characteristics, both horizontally and vertically (e.g., in an environment that also includes border routers), would give more insight in the capabilities of the implementation. A more advanced and flexible infrastructure (e.g., GÉANT and ExoGENI) could facilitate more complex topologies but also make rolling out and debugging harder, which need to be taken into consideration.
- The routing implementation of IRATI turned out to be incapable of coping with link failures. During our project we were not able to identify the source of this problem. Unless this problem is caused by an oversight in the configuration or an obvious code defect, The source code and commits could be analysed for the regression.
- During our experiments we have not looked into how congestion influences the flows in terms of routing and path decision making due to the problems caused by the link failure tests. When rerouting is functioning properly it would be interesting to test whether congestion and QoS-cubes are taken into account during routing decisions.

- Since time only allowed experimenting with one implementation we could not reflect and verify the behaviour against other implementations. In a future project multiple implementations could be compared with each other by means of experiments and code analysis.
- One deliverable of the PRISTINE project was the all-in-one-testing tool that allows one to quickly launch a multi-VM test environment from one base system and a topology configuration file. For this project we have chosen to not look into this because the indirection and/or abstraction could make setting up and debugging the test environment more difficult. Future research could be done into the capabilities and caveats of this tool in order to determine which use cases it is applicable to and to which extent it improves in the roll-out of a test environment.
- We found that the experimental Wireshark dissectors failed to correctly dissect the RINA related packets. This seemed to be caused by the configurable length of the fields and possibly a change to the packet format. We were able patch the EFCP dissector such that it correctly displays the source and destination IPCP addresses. When the dissectors are properly patched or enhanced, such that all the contents of the packets are displayed correctly, it would help the analysis of the flows to a great extent. A problem with the current dissectors is that in IRATI the fields are of configurable length while they are statically defined in the Wireshark dissectors. Research could be done into the possibilities of creating (dynamically) adjustable dissectors. Since Wireshark only can be connected to the bridges only traffic flowing over the physical links could be analysed. While the IRATI debugging features enable developers to inspect the functions internal to a machine, network engineers without comprehensive programming skills are likely unable to properly analyse the inner workings. Therefore future work on this topic could also include whether it is possible to use a packet sniffer to inspect the flows within a machine (e.g., flows between IPCPs of different DIFs).

Bibliography

[1] J. Day, *Patterns in Network Architecture, A Return to Fundamentals*. Pearson, 2008, 429 pp., ISBN: 978-0-13-225242-3 (cit. on pp. 5, 8, 10, 15, 31).

[2] About the Pouzin Society, Pouzin Society, [Online]. Available: http://pouzinsociety.org/about (visited on 01/07/2016) (cit. on p. 5).

[3] Research projects, Poezin Society, [Online]. Available: http://pouzinsociety.org/research/projects (visited on 01/07/2016) (cit. on pp. 5, 6, 17, 18).

[4] E. Grasa, E. Trouva, S. Bunch, P. DeWolf, and J. Day, "Developing a RINA prototype over UDP/IP using TINOS," in *Proceedings of the 7th International Conference on Future Internet Technologies*, ser. CFI '12, Seoul, Korea: ACM, 2012, pp. 31–36, ISBN: 978-1-4503-1690-3. DOI: 10.1145/2377310.2377321. [Online]. Available: http://doi.acm.org/10.1145/2377310.2377321 (cit. on pp. 6, 17).

[5] Y. Wang, I. Matta, and N. Akhtar, "Experimenting with routing policies using ProtoRINA over GENI, 2014 third geni research and educational experiment workshop," Computer Science Department, Boston University, 2014. [Online]. Available: https://www.geni.net/?p=3133 (visited on 01/06/2016) (cit. on p. 6).

[6] Y. Wang, N. Akhtar, and I. Matta, "Programming routing policies for video traffic, 2014 third geni research and educational experiment workshop," Computer Science Department, Boston University, 2014. [Online]. Available: https://www.geni.net/?p=3133 (visited on 01/06/2016) (cit. on p. 6).

[7] Y. Wang, F. Esposito, I. Matta, and J. Day, *Recursive internetworking architecture (rina), Boston university prototype*, Programming Manual, version 1.0, Boston University, Nov. 8, 2013, 43 pp. [Online]. Available: http://csr.bu.edu/rina/papers/BUCS-TR-2013-013.pdf (visited on 01/06/2016) (cit. on pp. 6, 17).

[8] V. Ishakian, J. Akinwumi, F. Esposito, and I. Matta, "On supporting mobility and multihoming in recursive internet architectures(2010)," *Technical Report BUCS-TR-2010-035*, 2010 (cit. on p. 6).

[9] S. Vrijders, D. Staessens, D. Colle, F. Salvestrini, E. Grasa, M. Tarzan, and L. Bergesio, "Prototyping the recursive internet architecture: the irati project approach," eng, *IEEE NETWORK*, vol. 28, no. 2, pp. 20–25, 2014, ISSN: 0890-8044 (cit. on p. 6).

[10] V. Llobet, E. Grasa, and S. Figuerola, "D1.5 final project report, Investigating RINA as an alternative to tcp/ip," IRATI, version 1.0, Feb. 19, 2015, 71 pp. [Online]. Available: http://irati.eu/wp-content/uploads/2012/07/IRATI-D1.5.pdf (visited on 01/06/2016) (cit. on p. 6).

[11] D. Staessens, "Final report on IRINA and software prototype (IRINA), Open call deliverable ocm-ds1.1," Géant, 2015, 123 pp. (cit. on pp. 6, 22, 24).

[12] V. Veselý, M. Marek, T. Hykel, and O. Ryšavý, "Rinasim: your recursive internetwork architecture simulator," presented at the OMNeT++ Community Summit 2015 (Zurich, Sep. 3, 2015), 7, 2015. [Online]. Available: https://summit.omnetpp.org/archive/2015/#keynotes (visited on 01/06/2016) (cit. on pp. 6, 13).

[13] "RINA simulator; basic functionality, Deliverable-2.4," PRISTINE, Jan. 31, 2015, 120 pp. [Online]. Available: http://ict-pristine.eu/wp-content/uploads/2013/12/pristine-d24-rinasim-v1_0.pdf (cit. on p. 6).

[14] Pouzin Society. (Dec. 22, 2015). New research project on large-scale RINA experimentation funded by the European Commission, [Online]. Available: http://www.pouzinsociety.org/news/arcfire (visited on 02/03/2016) (cit. on p. 6).

[15] J. Day, E. Trouva, E. Grasa, P. Phelan, M. P. D. Leon, S. Bunch, I. Matta, L. T. Chitkushev, and L. Pouzin, *Bounding the router table size in an ISP network using RINA*, 2011. DOI: 10.1109/NOF.2011.6126683 (cit. on p. 6).

[16] S. Knappstein-Hamelink, "Network discovery in a recursive internet network architecture," Technische Universiteit Delft, Feb. 26, 2014, 81 pp. [Online]. Available: http://repository.tudelft.nl/view/ir/uuid:9fedf129-104d-4401-9900-e4995b24eed1/ (visited on 01/23/2016) (cit. on p. 6).

[17] I. E. T. Force, "Requirements for internet hosts – communication layers," Internet Engineering Task Force, Tech. Rep. RFC1122, Oct. 1989. [Online]. Available: https://tools.ietf.org/html/rfc1122 (cit. on p. 7).

[18] F. Goldstein and J. Day, "Moving beyond TCP/IP," Apr. 2010. [Online]. Available: http://rina.tssg.org/docs/PSOC-MovingBeyondTCP.pdf (visited on 01/15/2016) (cit. on p. 7).

[19] J. Day, I. Matta, and K. Mattar, "Networking is IPC": a guiding principle to a better internet," in *in Proceedings of ReArch'08 - Re-Architecting the Internet*, 2008 (cit. on p. 8).

[20] F. Salvestrini, G. Carrozzo, P. Cruschelli, A. Chappel, J. Graham, S. Vrijders, D. Staessens, M. Tarzan, L. Bergesio, E. Trouva, E. Grasa, A. Vico, and C. Bermudo, "D2.1 first phase use cases, requirements analysis, RINA specifications and high-level software architecture, Investigating rina as an alternative to tcp/ip," IRATI, version V1.0, May 15, 2013, 225 pp. [Online]. Available: http://irati.eu/wp-content/uploads/2012/07/IRATI-D2.1.pdf (visited on 01/22/2016) (cit. on pp. 9, 14, 15, 19, 29).

[21] V. Veselý, M. Marek, T. Hykel, and O. Ryšavý, "Skip this paper - RINASim: your Recursive InterNetwork Architecture simulator," *CoRR*, vol. abs/1509.03550, 2015. [Online]. Available: http://ict-pristine.eu/wp-content/uploads/2013/12/pristine-d24-rinasim-v1_0.pdf (cit. on pp. 9, 18).

[22] Y. Wang, I. Matta, F. Esposito, and J. Day, "Introducing ProtoRINA: a prototype for programming recursive-networking policies," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 129–131, Jul. 2014, ISSN: 0146-4833. DOI: 10.1145/2656877.2656897. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656897 (visited on 01/19/2016) (cit. on pp. 11, 17).

[23] R. Watson, "The delta-t transport protocol: features and experience useful for high performance networks," in. Jan. 1989 (cit. on p. 11).

[24] F. S. Dimitri Staessens and M. Tarzan, *Future network architectures recursive internet architecture (RINA), Investigating RINA as an alternative to TCP/IP*. [Online]. Available: http://irati.eu/wp-content/uploads/2012/07/ISO-RINA.pdf (visited on 01/19/2016) (cit. on pp. 11, 17).

[25] J. H. Saltzer, "On the naming and binding of network destinations," *Local Computer Networks*, P. Ravasio *et al.*, Eds., pp. 311–317, 1982, republished as RFC 1498 (1993). [Online]. Available: http://tools.ietf.org/html/rfc1498 (cit. on p. 12).

[26] E. Grasa, E. Trouva, P. Phelan, M. P. de Leon, J. Day, I. Matta, L. T. Chitkushev, and S. Bunch, "Design principles of the recursive internetwork architecture (RINA)," 2011. [Online]. Available: http://www.future-internet.eu/fileadmin/documents/fiarch23may2011/06-Grasa_DesignPrinciplesOTheRecursiveInterNetworkArchitecture.pdf (visited on 01/29/2016) (cit. on p. 14).

[27] RINA Specs Wiki. (2012). The interina reference model, [Online]. Available: http://nes.fit.vutbr.cz/ivesely/specs/pmwiki.php/RINA/RefModelPart3 (visited on 01/22/2016), draft (cit. on p. 14).

[28] RINA Specs Wiki. (Dec. 28, 2014). Documents, Technical University in Brno (VUT), [Online]. Available: http://nes.fit.vutbr.cz/ivesely/specs/pmwiki.php/RINA/Documents (visited on 01/26/2016) (cit. on pp. 15, 47).

[29] "Pristine reference framework, Deliverable-2.2," PRISTINE, Jul. 30, 2014, 125 pp. [Online]. Available: http://ict-pristine.eu/wp-content/uploads/2013/12/pristine_d22-ref-framework_draft.pdf (cit. on pp. 15, 20).

[30] "Proof of concept software for the use cases and draft report on the use cases trials and business impact, Deliverable-6.2," PRISTINE, Sep. 31, 2015, 135 pp. [Online]. Available: http://ict-pristine.eu/wp-content/uploads/2013/12/pristine_d62-use-case-trials-and-business-impact_draft.pdf (cit. on pp. 15, 20, 27, 46, 70).

[31] IRATI. Learn: implementing and experimenting with RINA, IRATI, [Online]. Available: http://irati.eu/learn-implementing-rina/ (visited on 01/19/2016) (cit. on p. 17).

[32] Revert "added projects/rina and rina folder", GitHub, [Online]. Available: https://github.com/PouzinSociety/tinos/commit/66ca366a99f8ff086354abf32e7e7ff8534f3bde (visited on 01/19/2016) (cit. on p. 17).

[33] E. Grasa. (Dec. 1, 2011). Rina prototype, i2CAT Distributed Applications and Networks Area, [Online]. Available: http://dana.i2cat.net/?post_type=project&p=1384 (visited on 01/19/2016) (cit. on p. 17).

[34] S. Bunch, *TRIA implementation experience notes*, 16 2, 2015. [Online]. Available: http://ict-pristine.eu/wp-content/uploads/2014/12/TRIA-RINA-implementations.pdf (visited on 01/19/2016) (cit. on p. 17).

[35] J. Day, *Multiplexing and congestion control*, US Patent 8,180,918, May 2012. [Online]. Available: https://www.google.com/patents/US8180918 (cit. on p. 17).

[36] J. Day, *Parameterized recursive network architecture with topological addressing*, US Patent 8,352,587, Jan. 2013. [Online]. Available: https://www.google.com/patents/US8352587 (cit. on p. 17).

[37] J. Day and S. Bunch, *Method and system for managing network communications*, US Patent App. 14/211,928, Sep. 2014. [Online]. Available: https://www.google.com/patents/US20140269726 (cit. on p. 17).

[38] J. Day, *Parameterized recursive network architecture with topological addressing*, US Patent 8,769,077, Jul. 2014. [Online]. Available: https://www.google.com/patents/US8769077 (cit. on p. 17).

[39] D. D. Bock. (May 16, 2013). IRATI/stack), [Online]. Available: https://github.com/IRATI/stack/ (visited on 01/15/2016) (cit. on p. 18).

[40] D. D. Bock. (Feb. 2015). IRATI/stack wiki - getting started), [Online]. Available: https://github.com/IRATI/stack/wiki/Getting-Started (visited on 01/15/2016) (cit. on pp. 18, 57).

[41] R. Callon, "Use of OSI IS-IS for routing in TCP/IP and dual environments," Internet Engineering Task Force, Tech. Rep. RFC1195, Dec. 1990. [Online]. Available: https://tools.ietf.org/html/rfc1195 (cit. on p. 19).

[42] A. Atlas, "Basic specification for IP fast reroute: loop-free alternates," Internet Engineering Task Force, Tech. Rep. RFC5286, Sep. 2008. [Online]. Available: https://tools.ietf.org/html/rfc5286 (cit. on p. 20).

[43] D. Oran, "OSI IS-IS intra-domain routing protocol," Internet Engineering Task Force, Tech. Rep. RFC1142, Feb. 1990. [Online]. Available: http://tools.ietf.org/html/rfc1142 (cit. on p. 20).

[44] J. García. (Dec. 11, 2015). Pristine 1.3 ecmp implementation pull request #810, [Online]. Available: https://github.com/IRATI/stack/pull/810 (visited on 01/22/2016) (cit. on p. 20).

[45] E. Grasa. (Mar. 12, 2015). IRATI/stack wiki - tutorial 1: DIF over a VLAN (point to point dif), [Online]. Available: https://github.com/IRATI/stack/wiki/Tutorial-1:-DIF-over-a-VLAN-(point-to-point-DIF) (visited on 01/15/2016) (cit. on p. 22).

[46] Netdotnet. (Dec. 16, 2014). IRATI/stack wiki - tutorial 2: DIF over two VLANS, [Online]. Available: https://github.com/IRATI/stack/wiki/Tutorial-2:-DIF-over-two-VLANS (visited on 02/05/2016) (cit. on p. 25).

[47] E. Grasa. (Apr. 29, 2015). Ipc manager daemon: prevent assignment of two ipcps to the same dif in the same system, [Online]. Available: https://github.com/IRATI/stack/issues/558 (visited on 02/03/2016) (cit. on p. 27).

[48] "Draft conceptual and high-level engineering design of innovative security and reliability enablers, Deliverable-4.1," PRISTINE, Apr. 30, 2015, 127 pp. [Online]. Available: http://ict-pristine.eu/wp-content/uploads/2013/12/pristine_d41-security-and-reliability-enablers_draft.pdf (cit. on pp. 30, 40).

[49] RINA Specs Wiki. (Dec. 28, 2014). RINA/EFCP, Technical University in Brno (VUT), [Online]. Available: http://nes.fit.vutbr.cz/ivesely/specs/pmwiki.php/RINA/EFCP (visited on 02/06/2016) (cit. on p. 31).

Acronyms

A

maximum time before ACK 11

AE

Application Entity 8, 9

AP

Application Process 8–10, 12, 13, 35

CDAP

Common Distributed Application Protocol 11, 12, 22, 30, 57

CEP-id

connection-endpoint identifier 12

DAF

Distributed Application Facility 8, 9, 14

DAP

Distributed Application Process 8, 9

DIF

Distributed IPC Facility 6, 9–15, 17–22, 24, 25, 27, 29–31, 34, 36, 39, 40, 45, 46, 48, 49, 68, 75

DTCP

Data Transfer Control Protocol 11, 19

DTP

Data Transfer Protocol 11, 19

ECMP

equal-cost multipath routing 20

EFCP

Error and Flow Control Protocol 11, 12, 19, 22, 30, 44, 49, 57, 76, 78

EFCPM

Error and Flow Control Protocol Machine 12

FLD

Flow Liveness Detection 30

ICMP

Internet Control Message Protocol 34

IDD

Inter-DIF Directory 14, 20

IP

Internet Protocol 7, 12

IPC

inter-process communication 8–10

IPCM

inter-process communication manager 24, 30, 38, 45–47, 69, 75

IPCP

Inter-process Communication Process 9–13, 15, 19, 22, 24, 25, 27, 29–31, 34, 35, 39–41, 45, 46, 49, 75, 76

IRC

Internet Relay Chat 19

MPL

maximum packet lifetime 11

MSS

maximum segment size 36

MTU

maximum transmission unit 36, 37, 45, 59

NIC

network interface controller 7, 36, 37, 45, 46, 59

PCI

protocol control information 12

PDU

protocol data unit 12, 15, 30–32, 35, 37, 39, 41, 43, 44

QoS

quality of service 12, 15, 19, 48

R

maximum number of retries 11

RIB

resource information base 14, 15, 22, 25, 29, 35, 39, 41, 43, 45, 46, 76

RINA

Recursive InterNetwork Architecture ii, 5–9, 11, 12, 14, 15, 17–20, 22, 39, 45, 47–49

RTT

round-trip time 35

SDU

service data unit 11, 15, 18, 19, 22, 34–38, 40, 45

TCP

Transmission Control Protocol 11

UDP

User Datagram Protocol 6, 11

VLAN

virtual local area network 19, 22, 24, 25, 43, 59

VM

virtual machine 24, 25, 49, 57, 59, 65, 68, 70

A Test setup details

In this appendix more details of the test setup introduced in Chapter 4 are given.

A.1 KVM

`virt-manager` was used to install and operate the VMs. `qemu-img` was used to create a larger disk image than the graphical interface allows (the usage of the image can get close to 20 GiB due to the compilation of the kernel and since recompilation can be a lengthy process it was chosen to not clean the build directory). `qemu-img` was also used to create linked clones of the base system.

A.2 IRATI stack

The IRATI stack was installed according to the directions given on the IRATI GitHub wiki [40] (some minor deviations were necessary to accommodate for the differences between Debian 7 and Debian 8).

The following packages were installed (all from the Debian repository): `kernel-package libncurses5-dev autoconf automake libtool pkg-config git g++ openjdk-7-jdk maven protobuf-compiler libprotobuf-dev libnl-3-dev swig libssl-dev`

The IRATI stack was obtained from `https://github.com/IRATI/traffic-generator.git`

The kernel was compiled manually using the commands supplied by the IRATI wiki to be able to exert more control over the compilation process (e.g., specify the amount of build jobs) instead of using the supplied script. The following kernel options were disabled: `RINA debugging support` (comprising the disablement of `dump heartbeat messages` and `embed assertions`). (See Table A.1 for a detailed listing of the resulting options.)

The IRATI user-space was compiled using the `install-user-from-scratch` script and installed into the `/usr/local/irati` prefix.

A.3 RINA traffic generator & Wireshark

In order to test the performance of the test setup the RINA traffic generator was used. It was obtained from `https://github.com/IRATI/traffic-generator.git`. The following additional package was installed: `libboost-all-dev`.

In order to inspect the network packets the RINA-enabled version of Wirehark was obtained from `https://github.com/IRATI/wireshark.git`. To compile it the following additional packages were installed: `libpcap-dev bison flex libtool-bin glib-2.0 libgtk2.0-dev`. In order to compile the CDAP & EFCP disector plugins, it is important to run the `autogen.sh` script prior to compilation.

Enabled options
CONFIG_RINA=y
CONFIG_RINA_NORMAL_IPCP=m
CONFIG_RINA_DTCP_RCVR_ACK=y
CONFIG_RINA_SHIM_ETH_VLAN=m
CONFIG_RINA_SHIM_TCP_UDP=m
CONFIG_RINA_SHIM_TCP_UDP_BUFFER_SIZE=1500
CONFIG_RINARP=m
Disabled options
CONFIG_RINA_DEBUG
CONFIG_RINA_DEBUG_HEARTBEATS
CONFIG_RINA_ASSERTIONS
Unset options
CONFIG_RINA_DTCP_RCVR_ACK_ATIMER
CONFIG_RINA_SHIM_ETH_VLAN_BURST_LIMITING
CONFIG_RINA_SHIM_ETH_VLAN_REGRESSION_TESTS

Table A.1: RINA kernel configuration

B Test scenario configuration details

In this appendix the configuration details of the test scenarios are given.

B.1 Scenario 1: basic tests

B.1.1 Generic configuration

For the bandwidth tests the MTU has been adjusted. Only changing the MTU in the VMs was not sufficient; also the MTU of the virtual NICs outside the VMs on the host needed to be changed.

The VMs were assigned two cores 2048 MB RAM. Both had internet reachability at eth0 while eth1 was attached to an isolated bridge that connected the two VMs.

B.1.2 System 1

As stated in Chapter 4 the first test scenario is based on the first IRATI tutorial. However, we have chosen to change the VLAN from 100 to 110 since this is the default configuration that the stack is shipped with. The reasoning behind this is that this way the tests are easier to reproduce. Therefore for the following configuration files are given solely for the sake of completeness.

```
{
  "configFileVersion" : "1.4.1",
  "localConfiguration" : {
    "installationPath" : "/usr/local/irati/bin",
    "libraryPath" : "/usr/local/irati/lib",
    "logPath" : "/usr/local/irati/var/log",
    "consolePort" : 32766,
    "pluginsPaths" : ["/usr/local/irati/lib/rinad/ipcp"]
  },
  "ipcProcessesToCreate" : [ {
    "type" : "shim-eth-vlan",
    "apName" : "test-eth-vlan",
    "apInstance" : "1",
    "difName" : "110"
   }, {
    "type" : "normal-ipc",
    "apName" : "test1.IRATI",
    "apInstance" : "1",
    "difName" : "normal.DIF",
    "difsToRegisterAt" : ["110"]
```

```
   } ],
  "difConfigurations" : [ {
    "name" : "110",
    "template" : "shim-eth-vlan.dif"
  }, {
    "name" : "normal.DIF",
    "template" : "default.dif"
  } ]
}
```

Listing 22: ipcmanager.conf

```
{
    "difType" : "shim-eth-vlan",
    "configParameters" : {
            "interface-name" : "eth1"
      }
}
```

Listing 23: shim-eth-vlan.dif

```
{
    "difType" : "normal-ipc",
    "dataTransferConstants" : {
            "addressLength" : 2,
            "cepIdLength" : 2,
            "lengthLength" : 2,
            "portIdLength" : 2,
            "qosIdLength" : 2,
            "rateLength" : 4,
            "frameLength" : 4,
            "sequenceNumberLength" : 4,
            "ctrlSequenceNumberLength" : 4,
            "maxPduSize" : 10000,
            "maxPduLifetime" : 60000
    },
    "qosCubes" : [ {
         "name" : "unreliablewithflowcontrol",
         "id" : 1,
         "partialDelivery" : false,
         "orderedDelivery" : true,
         "efcpPolicies" : {
              "dtpPolicySet" : {
                "name" : "default",
```

```
            "version" : "0"
          },
          "initialATimer" : 300,
          "dtcpPresent" : true,
          "dtcpConfiguration" : {
               "dtcpPolicySet" : {
                 "name" : "default",
                 "version" : "0"
               },
               "rtxControl" : false,
               "flowControl" : true,
               "flowControlConfig" : {
                   "rateBased" : false,
                   "windowBased" : true,
                   "windowBasedConfig" : {
                     "maxClosedWindowQueueLength" : 50,
                     "initialCredit" : 50
                    }
               }
          }
     }
  }, {
          "name" : "reliablewithflowcontrol",
    "id" : 2,
    "partialDelivery" : false,
    "orderedDelivery" : true,
    "maxAllowableGap": 0,
    "efcpPolicies" : {
          "dtpPolicySet" : {
            "name" : "default",
            "version" : "0"
          },
          "initialATimer" : 300,
          "dtcpPresent" : true,
          "dtcpConfiguration" : {
               "dtcpPolicySet" : {
                 "name" : "default",
                 "version" : "0"
               },
               "rtxControl" : true,
               "rtxControlConfig" : {
                   "dataRxmsNmax" : 5,
                   "initialRtxTime" : 1000
               },
               "flowControl" : true,
               "flowControlConfig" : {
                   "rateBased" : false,
```

```
                    "windowBased" : true,
                    "windowBasedConfig" : {
                      "maxClosedWindowQueueLength" : 50,
                      "initialCredit" : 50
                              }
                         }
                  }
     }
  } ],
  "knownIPCProcessAddresses" : [ {
          "apName" : "test1.IRATI",
          "apInstance" : "1",
          "address" : 16
         }, {
          "apName" : "test2.IRATI",
          "apInstance" : "1",
          "address" : 17
       } ],
       "addressPrefixes" : [ {
          "addressPrefix" : 0,
          "organization" : "N.Bourbaki"
         }, {
          "addressPrefix" : 16,
          "organization" : "IRATI"
   } ],
  "rmtConfiguration" : {
     "pffConfiguration" : {
       "policySet" : {
         "name" : "default",
         "version" : "0"
       }
     },
     "policySet" : {
       "name" : "default",
       "version" : "1"
     }
  },
  "enrollmentTaskConfiguration" : {
     "policySet" : {
        "name" : "default",
        "version" : "1",
        "parameters" : [{
            "name"  : "enrollTimeoutInMs",
            "value" : "10000"
          },{
            "name"  : "watchdogPeriodInMs",
            "value" : "30000"
```

```
            },{
              "name"  : "declaredDeadIntervalInMs",
              "value" : "120000"
            },{
              "name"  : "neighborsEnrollerPeriodInMs",
              "value" : "30000"
            },{
              "name"  : "maxEnrollmentRetries",
              "value" : "3"
            }]
       }
    },
    "flowAllocatorConfiguration" : {
        "policySet" : {
          "name" : "default",
          "version" : "1"
         }
    },
    "namespaceManagerConfiguration" : {
        "policySet" : {
          "name" : "default",
          "version" : "1"
          }
    },
    "securityManagerConfiguration" : {
        "policySet" : {
          "name" : "default",
          "version" : "1"
          }
    },
    "resourceAllocatorConfiguration" : {
        "pduftgConfiguration" : {
          "policySet" : {
            "name" : "default",
            "version" : "0"
          }
        }
    },
    "routingConfiguration" : {
        "policySet" : {
          "name" : "link-state",
          "version" : "1",
          "parameters" : [{
            "name"  : "objectMaximumAge",
            "value" : "10000"
          },{
            "name"  : "waitUntilReadCDAP",
```

```
            "value" : "5001"
          },{
            "name"  : "waitUntilError",
            "value" : "5001"
          },{
            "name"  : "waitUntilPDUFTComputation",
            "value" : "103"
          },{
            "name"  : "waitUntilFSODBPropagation",
            "value" : "101"
          },{
            "name"  : "waitUntilAgeIncrement",
            "value" : "997"
          },{
            "name"  : "routingAlgorithm",
            "value" : "Dijkstra"
          }]
    }
  }
}
```

Listing 24: default.dif

```
{
  "applicationToDIFMappings" : [ {
    "encodedAppName" : "traffic.generator.server-1--",
    "difName" : "normal.DIF"
  }, {
    "encodedAppName" : "traffic.generator.client-1--",
    "difName" : "normal.DIF"
  }, {
    "encodedAppName" : "rina.apps.echotime.server-1--",
    "difName" : "normal.DIF"
  }, {
    "encodedAppName" : "rina.apps.echotime.client-1--",
    "difName" : "normal.DIF"
  }]
}
```

Listing 25: da.map

B.1.3 System 2

System two requires one changed line. In `ipcmanager.conf` the line `"apName" : "test1.IRATI"` needs to be changed to `"apName" : "test2.IRATI"`.

B.1.4 Enrolment

Enrolment was done from system 1 and involved the following command (See Listing 26).

```
enroll-to-dif 2 normal.DIF 110 test2.IRATI 1    # System 2 via VLAN 101
```

Listing 26: Enrolment of system 2 to system 1

B.2 Scenario 2: routing tests

B.2.1 Generic configuration

The VMs were assigned 1024 MB RAM. System 1 and system 4 were assigned two cores since the applications were mainly run on system 1 and 4, and the bandwidth tests could be processor bound. the other systems had 1 core assigned. All the machines had internet reachability at eth0 while eth1 and eth2 were attached to isolated bridges. Because the designed setup uses interfaces eth1 through eth4 for these links `ifrename` was used where to change the name of the interfaces where necessary (other solutions could be used but this approach turned out to be the most flexible).

B.2.2 System 1

Test scenario 1 was used as the basis for the settings of scenario 2. The `ipcmanager and shim-dif` configuration files will be shown as a whole while the `default.dif` file will only contain the changes between scenario 1 and 2.

```
{
  "configFileVersion" : "1.4.1",
  "localConfiguration" : {
    "installationPath" : "/usr/local/irati/bin",
    "libraryPath" : "/usr/local/irati/lib",
    "logPath" : "/usr/local/irati/var/log",
    "consolePort" : 32766,
    "pluginsPaths" : ["/usr/local/irati/lib/rinad/ipcp"]
  },
  "ipcProcessesToCreate" : [ {
    "type" : "shim-eth-vlan",
    "apName" : "test-eth-vlan",
    "apInstance" : "1",
    "difName" : "101"
   },
  {
    "type" : "shim-eth-vlan",
    "apName" : "test-eth-vlan2",
```

```
    "apInstance" : "1",
    "difName" : "102"
   },
 {
    "type" : "normal-ipc",
    "apName" : "IPCP1.IRATI",
    "apInstance" : "1",
    "difName" : "normal.DIF",
    "difsToRegisterAt" : ["101", "102"]
   } ],
  "difConfigurations" : [ {
    "name" : "101",
    "template" : "shim-eth-vlan.dif"
  },
  {
    "name" : "102",
    "template" : "shim2-eth-vlan.dif"
  },
  {
    "name" : "normal.DIF",
    "template" : "default.dif"
  } ]
}
```

Listing 27: ipcmanager.conf

```
{
    "difType" : "shim-eth-vlan",
    "configParameters" : {
            "interface-name" : "eth1"
      }
}
```

Listing 28: shim-eth-vlan.dif

```
{
    "difType" : "shim-eth-vlan",
    "configParameters" : {
            "interface-name" : "eth2"
      }
}
```

Listing 29: shim2-eth-vlan.dif

```
{[...]
     "knownIPCProcessAddresses" : [ {
     "apName" : "IPCP1.IRATI",
     "apInstance" : "1",
     "address" : 16
     }, {
     "apName" : "IPCP2.IRATI",
     "apInstance" : "1",
     "address" : 17
     }, {
     "apName" : "IPCP3.IRATI",
     "apInstance" : "1",
     "address" : 18
     }, {
     "apName" : "IPCP4.IRATI",
     "apInstance" : "1",
     "address" : 19
     } ],
 [...]}
```

Listing 30: changes to default.dif

B.2.3 Other systems

The changes that need to be made for the different systems are shown in Table B.1.

	System 1	**System 2**	**System 3**	**System 4**
shim-eth-vlan difName	101	101	102	103
configParameters interface-name	eth1	eth1	eth2	eth3
shim-eth-vlan2 difName	102	103	104	104
configParameters interface-name	eth2	eth3	eth4	eth4
normal-ipc apName	IPCP1.IRATI	IPCP2.IRATI	IPCP3.IRATI	IPCP4.IRATI
normal-ipc difsToRegisterAt	101, 102	101, 103	102, 104	103, 104

Table B.1: Configuration settings of individual systems

B.2.4 Enrolment

Enrolment was done from System 1 and System 4. On System 1 this involved the following commands (see Listing 31), while on System 4 the following commands were issued (see Listing 32). After these commands are run all the shim DIFs are enrolled and the network is for experiments.

```
enroll-to-dif 3 normal.DIF 101 IPCP2.IRATI 1   # System 2 via VLAN 101
enroll-to-dif 3 normal.DIF 102 IPCP3.IRATI 1   # System 3 via VLAN 102
```

Listing 31: Enrolment of system 2 and 3 to system 1

```
enroll-to-dif 3 normal.DIF 103 IPCP2.IRATI 1   # System 2 via VLAN 103
enroll-to-dif 3 normal.DIF 104 IPCP3.IRATI 1   # System 3 via VLAN 104
```

Listing 32: Enrolment of system 2 and 3 to system 4

B.2.5 Link failure

Part of the experiments for the second scenario was simulating link failure and observing the behaviour of the flows. We used several approaches to simulate the failure of links:

- Shut down the link within the VM
- Shut down the link of the guest on the bridge on the host
- Filter the traffic on the bridge

We have tried all approaches and found the last method to work the best. It has the least chance of disturbing the IRATI stack (there might be a remote chance that bringing down the link could lead to unexpected behaviour) and does not cause packet sniffers to raise errors and stop their capture because a link is brought down (which happened when a link from the bridge was shut down). It could also be useful in case one uses a single bridge to transfer multiple VLANS. We used the following commands to filter the traffic on the bridge (see Listing 33).

```
# Disable the link between System 1 and System 2.
ebtables -t broute -A BROUTING -p 802_1Q --vlan-id 101 -j DROP
# Re-enable the link between System 1 and System 2.
ebtables -t broute -F
```

Listing 33: Enrolment of system 2 and 3 to system 4

B.2.6 Multipath

The multipath plugin, included in the pristine 1.3 and 1.4 branches, is not installed by default. To install it `make install` needs to be run inside the `stack/plugins/multipath` folder. Then in `ipcmanager.conf` the `pluginPaths` member needs to hold the right `/lib/modules/$(KREL)/extra` variable.

```
{"pluginsPaths" : ["/usr/local/irati/lib/rinad/ipcp",
↪  "/lib/modules/4.1.10/extra"]}
```

Listing 34: changes to ipcmanager.conf

In the `default.dif` two changes need to be made to the routing policies. First the `pftConfiguration policyset` needs to be changed from `default` to `multipath`. Next in the `routingConfiguration` the `routingAlgorithm` member needs its value to change from `Dijkstra` to `ECMPDijkstra`

```
{[...]
        "pffConfiguration" : {
                "policySet" : {
                        "name" : "multipath",
                        "version" : "0"
                        }
                },
        [...]
        "routingConfiguration" : {
             "policySet" : {
             "name" : "link-state",
             "version" : "1",
             "parameters" : [{
                         [...],{
                    "name"  : "routingAlgorithm",
                    "value" : "ECMPDijkstra"
                    }]
             }
    }
[...]}
```

Listing 35: changes to ipcmanager.conf

The `pff-multipath` kernel module was manually loaded prior to starting the IPCM.

B.2.7 Alternative setup: Loop Free Alternates

After problems had arisen with regards to the routing resiliency, explanations and solutions were sought. After being unsuccessful with the multipath plugin a change to the topology

has been made in order to conduct the "Loop Free Alternates in RINA" experiment from Deliverable 6.2 of the pristine project.[30, p. 29]

In Listing 36 the `default.dif` configuration file is given that was used during this experiment. The other configuration files that had to be changed and the changes necessary to the setup of the VMs are not given due to these being trivial adaptations of the second test setup.

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

```
                }
            }
        }
    }, {
            "name" : "reliablewithflowcontrol",
      "id" : 2,
      "partialDelivery" : false,
      "orderedDelivery" : false,
      "maxAllowableGap": 0,
      "efcpPolicies" : {
            "dtpPolicySet" : {
              "name" : "default",
              "version" : "0"
            },
            "initialATimer" : 300,
            "dtcpPresent" : false,
            "dtcpConfiguration" : {
                "dtcpPolicySet" : {
                  "name" : "default",
                  "version" : "0"
                },
                "rtxControl" : true,
                "rtxControlConfig" : {
                    "dataRxmsNmax" : 5,
                    "initialRtxTime" : 1000
                },
                "flowControl" : false,
                "flowControlConfig" : {
                    "rateBased" : false,
                    "windowBased" : true,
                    "windowBasedConfig" : {
                      "maxClosedWindowQueueLength" : 50,
                      "initialCredit" : 50
                            }
                        }
                  }
    }
} ],
"knownIPCProcessAddresses" : [ {
        "apName" : "IPCP1.IRATI",
        "apInstance" : "1",
        "address" : 16
       }, {
        "apName" : "IPCP2.IRATI",
        "apInstance" : "1",
        "address" : 17
     }, {
```

```
        "apName" : "IPCP3.IRATI",
        "apInstance" : "1",
        "address" : 18
         }, {
        "apName" : "IPCP4.IRATI",
        "apInstance" : "1",
        "address" : 19
      } ],
        "addressPrefixes" : [ {
           "addressPrefix" : 0,
           "organization" : "N.Bourbaki"
         }, {
           "addressPrefix" : 16,
           "organization" : "IRATI"
   } ],
   "rmtConfiguration" : {
      "pffConfiguration" : {
        "policySet" : {
          "name" : "default",
          "version" : "0"
        }
      },
      "policySet" : {
        "name" : "default",
        "version" : "1"
      }
   },
   "enrollmentTaskConfiguration" : {
      "policySet" : {
         "name" : "default",
         "version" : "1",
         "parameters" : [{
             "name"  : "enrollTimeoutInMs",
             "value" : "10000"
          },{
             "name"  : "watchdogPeriodInMs",
             "value" : "5000"
          },{
             "name"  : "declaredDeadIntervalInMs",
             "value" : "15000"
          },{
             "name"  : "neighborsEnrollerPeriodInMs",
             "value" : "30000"
          },{
             "name"  : "maxEnrollmentRetries",
             "value" : "3"
          }]
```

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

```
            },{
              "name"  : "waitUntilAgeIncrement",
              "value" : "997"
            },{
              "name"  : "routingAlgorithm",
              "value" : "Dijkstra"
            }]
    }
  }
}
```

Listing 36: Alternative configuration file default.dif

C Workarounds for usability problems

During the experimentation we had some usability problems with certain IRATI tools. In this appendix workarounds are given for certain usability problems.

C.1 Console commands

The IPCM console proved difficult to interact with and to be able to work around the problems this caused we used the commands below.

Scripting the enrolment of remote IPCPs to a local DIF can be fairly easily done by using a here document as input to the Telnet command (see Listing 37). Unfortunately this does not show the output of the IPCM console so the IPCM log output should be consulted to check whether the command was successful (see Listing 38).

```
#!/bin/bash
telnet localhost 32766 <<EOF
enroll-to-dif 2 normal.DIF 110 test2.IRATI 1
exit
EOF
```

Listing 37: Bash script to easily enrol a remote IPCP to a local DIF

```
# ./rina-enrol
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
IPCM >>> Connection closed by foreign host.
```

Listing 38: The actual output of the IPCM is not shown]

Because the before-mentioned Telnet trick does not work when console output is required netcat was used to wrap certain IPCM console commands in (see Listings 39 and 40).

```
nc -w 1 localhost 32766 <<< "list-ipcps"
```

Listing 39: Netcat command to easily list the IPCPs

```
nc -w 1 localhost 32766 <<< "query-rib 3"
```

Listing 40: Netcat command to easily query the RIB of an IPCP

To further improve on this we created aliases so that these snippets are executable like normal system commands (see Listing 41).

```
query-rib-function() {
  if [[ -n $1 ]]; then
    var=$1
  else
    var=3        # default ipcp to query the rib from
  fi
  nc -w 1 localhost 32766 <<< "query-rib $var"
}

alias list-ipcps='nc -w 1 localhost 32766 <<< "list-ipcps"'
alias query-rib='query-rib-function'
```

Listing 41: Aliases for netcat commands to easily query the RIB of an IPCP

C.2 Wireshark patch

The RINA-enabled Wireshark proved unable to properly dissect the EFCP packets. After consulting the developer it was stated that this was possibly caused by a difference between the configurable field lengths in IRATI and the (at compile time) fixed lengths as defined in the Wireshark Dissector. After changing the field lengths in IRATI did not result in a noticeable improvement the fields were changed in Wireshark dissector. With the knowledge that the fields are little-endian encoded (`packet-efcp.c` states `ENC_LITTLE_ENDIAN` as encoding) and that the address fields are two bytes long we figured out that the `pdu_type` probably needs to be 1 byte long and that it currently overlaps with the value `1200` which happens to be 18 in hexadecimal and seems to correctly correspond to the `dst_addr`. The next field then lists the value `1000` which is 16 in hexadecimal and correctly corresponds to the `src_addr`.

Using this approach we have come up with a patch (see Listing 42) that appears to correctly display at least some of the fields. Unfortunately there are still fields that are likely to be wrong and the packets that originate from applications still show up as malformed (see Listing 43 for an example). We have not put much effort in this patch and think it should be trivial to improve upon such that all the fields can be properly dissected (but unfortunately we lacked the time to do this).

After applying the patch, the following can be said about the correctness of the fields:

- PDU Type; always one, possibly wrong.
- Destination address, Source Address; verified to be correct.

- Destination Connection Endpoint ID, Source Connection Endpoint ID, Quality of Service ID; appear to be correct, but do fluctuate a bit thus are suspected to be wrong.
- PDU flags and the rest of the fields; still fluctuate heavily and these fields need to be analysed more to so that the offsets can be set correctly.

```
--- packet-efcp.c        2016-01-26 17:19:56.120355418 +0100
+++ packet-efcp.c        2016-01-26 17:22:09.745448594 +0100
@@ -61,26 +61,26 @@
"Unknown (0x%02x)"));
rina_tree = proto_item_add_subtree(ti, ett_rina);
proto_tree_add_item(rina_tree, hf_rina_pdu_type,
-                                    tvb, offset, 4, ENC_LITTLE_ENDIAN);
-                offset += 4;
+                                    tvb, offset, 1, ENC_LITTLE_ENDIAN);
+                offset += 1;
proto_tree_add_item(rina_tree, hf_rina_dst_addr,
-                                    tvb, offset, 4, ENC_LITTLE_ENDIAN);
-                offset += 4;
+                                    tvb, offset, 2, ENC_LITTLE_ENDIAN);
+                offset += 2;
proto_tree_add_item(rina_tree, hf_rina_src_addr,
-                                    tvb, offset, 4, ENC_LITTLE_ENDIAN);
-                offset += 4;
+                                    tvb, offset, 2, ENC_LITTLE_ENDIAN);
+                offset += 2;
proto_tree_add_item(rina_tree, hf_rina_dst_cep,
-                                    tvb, offset, 4, ENC_LITTLE_ENDIAN);
-                offset += 4;
+                                    tvb, offset, 2, ENC_LITTLE_ENDIAN);
+                offset += 2;
proto_tree_add_item(rina_tree, hf_rina_src_cep,
-                                    tvb, offset, 4, ENC_LITTLE_ENDIAN);
-                offset += 4;
+                                    tvb, offset, 2, ENC_LITTLE_ENDIAN);
+                offset += 2;
proto_tree_add_item(rina_tree, hf_rina_qos_id,
-                                    tvb, offset, 4, ENC_LITTLE_ENDIAN);
-                offset += 4;
+                                    tvb, offset, 2, ENC_LITTLE_ENDIAN);
+                offset += 2;
proto_tree_add_item(rina_tree, hf_rina_pdu_flags,
-                                    tvb, offset, 4, ENC_LITTLE_ENDIAN);
-                offset += 4;
+                                    tvb, offset, 1, ENC_LITTLE_ENDIAN);
+                offset += 1;
proto_tree_add_item(rina_tree, hf_rina_seq_num,
```

```
tvb, offset, 4, ENC_LITTLE_ENDIAN);
offset += 4;
```

Listing 42: Wireshark EFCP dissector patch

```
$ tshark -r Dump1.pcapng -Y "frame.number == 30" -O efcp -x
Frame 30: 57 bytes on wire (456 bits), 57 bytes captured (456 bits) on
 ↪  interface 1
Ethernet II, Src: RealtekU_68:57:26 (52:54:00:68:57:26), Dst:
 ↪  RealtekU_38:72:65 (52:54:00:38:72:65)
802.1Q Virtual LAN, PRI: 0, CFI: 0, ID: 102
Error and Flow Control Protocol, Unknown (0x1201) PDU
PDU Type: Unknown (0x00000001)
 Destination address: 18
 Source address: 16
Destination Connection Endpoint ID: 1
Source Connection Endpoint ID: 0
Quality of Service ID: 0
PDU Flags: 128
Sequence number: 2768250752
ACK/NACK sequence number: 342208804
New right window edge: 404166165
New left window edge: 471538201
Left window edge: 538910237
Right window edge: 606282273
[Malformed Packet: EFCP]

0000  52 54 00 38 72 65 52 54 00 68 57 26 81 00 00 66   RT.8reRT.hW&...f
0010  d1 f0 01 12 00 10 00 01 00 00 00 00 00 80 80 27   ...............'
0020  00 a5 24 b1 65 14 15 16 17 18 19 1a 1b 1c 1d 1e   ..$.e...........
0030  1f 20 21 22 23 24 25 26 27                        . !"#$%&'
```

Listing 43: Output of Wireshark using the patched dissector