University of Amsterdam

University of Amsterdam
Master System and Network Engineering
Research Project

Taking a closer look at IRATI

Student:
Koen Veelenturf
Koen.Veelenturf@os3.nl

Supervisor(s):
Marijke Kaat
Marijke.Kaat@surfnet.nl

Ralph Koning
R.Koning@uva.nl

July 20, 2016

Abstract

The Recursive InterNetwork Architecture (RINA) is a new Internet architecture that tries to solve many of the current Internet's problems. In this report, the IRATI implementation has been assessed on the topic of multihoming. Currently, it is not fully possible to use the multihoming capabilities of this RINA implementation, because the mapping between applications and DIFs is still static. However, there is active development in this field and in the upcoming two years, projects will research and further develop the multihoming and mobility capabilities.

Acknowledgements

I would like to express my special thanks and gratitude to my supervisors Marijke Kaat (SURFnet) and Ralph Koning (UvA), who gave me the opportunity to do this research. They helped me a lot during my research by making time for me and giving me helpful feedback. Secondly, I'd like to thank Jeroen Klomp and Jeroen van Leur for helping me with some problems with the IRATI stack and Guido Kroon, Wouter Miltenburg, and Stella Vouteva for giving helpful feedback on my report. Finally, I would like to thank Eduard Grasa Gras and Vincenzo Maffione (IRATI Project) for their technical support.

Contents

1 Introduction **4**
1.1 Research Question . . . 5
1.2 Related Work . . . 5

2 The Multihoming Problem **6**
2.1 A bit of Internet History . . . 6
2.2 Definition of Multihoming . . . 8
2.3 Current solutions to the multihoming problem . . . 9
2.3.1 Multihoming with IPv4 . . . 9
2.3.2 Multihoming with IPv6 . . . 10
2.3.3 Other solutions for the multihoming problem . . . 11

3 The Recursive InterNetwork Architecture **13**
3.1 Background Information . . . 13
3.2 RINA and Multihoming . . . 15
3.3 Related RINA Projects . . . 16
3.3.1 IRATI . . . 16
3.3.2 PRISTINE . . . 16
3.3.3 ARCFIRE . . . 17

4 Experiments **18**
4.1 Test Environment . . . 18
4.2 Basic tests with IRATI . . . 19
4.3 Multihoming with IRATI . . . 20
4.3.1 Test cases . . . 21
4.3.2 IRATI Demonstrator . . . 22
4.3.3 IRATI stack . . . 26

5 Discussion **31**
5.1 IRATI Demonstrator . . . 31
5.2 Compiled IRATI stack . . . 31

6 Conclusion **33**

7 Future Work **34**

8 Bibliography **35**

Glossary 40

Appendices 42

Appendix A Compiling and Installing the IRATI stack i
Appendix B Configuration Experiment #1: IRATI Demonstrator iii
Appendix C Configuration Experiment #2: VMs with compiled IRATI stack v
Appendix D Enrolling DIFs xiv
Appendix E Memory Script xvii

Chapter 1

Introduction

The Internet is based on the architecture of the ARPANET, which has evolved into a network architecture based on the TCP/IP stack. Throughout the years, quite some limitations were discovered in the current Internet architecture and implementation. One of the problems is that networks do not have a notion of node and/or application names. Therefore, applications need to have a combination of the *interface* address and the (transport layer) port number to identify different nodes and their services in a network. Every time a host's location changes its point-of-attachment, the logical location changes from a network point-of-view. This results in further complicating multihoming, mobility, and security.
Multihoming *is the practise of connecting a host or a computer network to more than one network* [60]. Typically, a host is connected to just one network. However, in many circumstances, it could be very useful to connect a host to multiple networks, e.g. for redundancy. Mobility is the practise of *changing a single unit's point of attachment to the Internet and therefore its reachability in the Internet topology* [14]. Multihoming and mobility are therefore closely related, but with multihoming you do not necessarily change the point-of-attachment as many times as one might with mobility.
Because the interfaces in the Internet are named and not the node and/or application, multihoming and mobility are very hard to achieve. Applications cannot handle multihoming very well out-of-the-box and therefore, special protocols and point solutions are necessary. The Recursive InterNetwork Architecture (RINA) is an effort of redesigning the Internet to solve multiple current Internet problems, including the multihoming problem. RINA is a specific architecture, implementation, testing platform, and ultimately, deployment of the theory, namely the Inter-Process Communication (IPC) model [9]. Furthermore, the IPC model also deals with concepts that are generic for distributed applications and therefore it is not only limited to networking.

1.1 Research Question

The research is formed around the main research question:
How does RINA solve the multihoming problem?

As a result of the main research question, the following sub research questions are created:

- What are the problems with multihoming and mobility in the current Internet?
- What kind of solutions are proposed to solve the multihoming/mobility problem in the current Internet?
- To what extend is multihoming/mobility implemented in the IRATI implementation?

1.2 Related Work

The principles behind RINA were first described in the book *Patterns in Network Architecture: A return to Fundamentals* by John Day [9]. Day takes the lessons learned of the almost forty years of TCP/IP experience, the lessons of OSI's failures, and other network technologies. John Day, et al. presented their ideas also during the CoNEXT conference in 2008 [13].
A paper by Grasa, et al. describes the design principles of the Recursive InterNetwork Architecture (RINA) [25]. Furthermore, the document describes the fundamental limitations of the current Internet and the path to the *future Internet*.
One of the implementations of the Recursive InterNetwork Architecture is IRATI Project [58]. Since 2014, they have been developing a prototype implementation of the Recursive InterNetwork Architecture. The IRATI implementation has a rich set of features for experimental testing of RINA.

Chapter 2

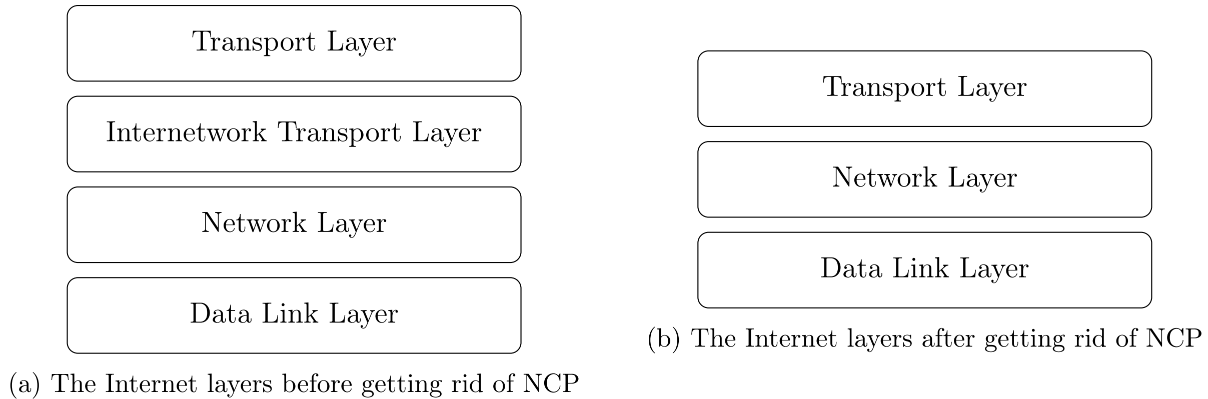
The Multihoming Problem

When the Sputnik 1 was launched by the Soviet Union in 1957, it lead to the creation of the Advanced Research Projects Agency (ARPA) in 1958. ARPA was later renamed into Defence ARPA (DARPA) [38]. (D)ARPA made plans for creating the first wide-area packet switching network, the ARPANET. The ARPANET is basically the basis for the current Internet. The almost fifty years of Internet evolution has led to several interesting challenges, problems, and solutions.

2.1 A bit of Internet History

In 1972, the researchers of ARPANET, NPL[1], CYCLADES[2], and other computer researchers decided that it would be a good idea to form an International Network Working Group (INWG). This lead to its major project, creating an international network transport protocol [41] so all the different networks running their own protocols could be interconnected to each other. In 1976 the INWG voted on an international transport protocol and the selected option had an architecture composed of three layers. A Data Link layer to handle different types of physical media, a Network Layer to handle different kind of networks, and an Internetwork Layer to handle a network of networks[3]. Each of these layers had their own address space. These layers are depicted in Figure 2.1a. When TCP/IP was introduced, it ran at the Internetwork Layer on top of the Network Control Program (NCP) and other network technologies. On the 1st of January 1983, also known as *flag day*, the ARPANET migrated into a full TCP/IP network, permanently activating TCP/IP and switching off NCP. This resulted in the disappearance of the Internetwork Transport Layer, as shown in Figure 2.1b.
Initially, early TCP versions performed error and flow control[4] and relaying and multiplexing functions in the same protocol. In 1978, Cerf, Postel, and Dany Cohen at ISI decided to split TCP into two separate functions of TCP and the Internet Protocol [53] [28]. This resulted in TCP being in charge of the creation of TCP segments and reassembling them at the destination side and IP is responsible for transmitting individual segments. Splitting the layers would not have been a problem if the two layers were independent and if the two layers did not contain repeated functions. However, these layers are dependent on each other and contain repeated functions. The Internet

[1] National Physical Laboratory Network, a British research network created in the 1970s
[2] A French research network created in the 1970s
[3] A network of networks is the concept of connecting separate networks to each other, creating one big network
[4] These functions are still in today's version of TCP



(a) The Internet layers before getting rid of NCP

(b) The Internet layers after getting rid of NCP

Figure 2.1: The different Internet layers

Protocol needs to be aware of what the Transmission Control Protocol is doing. Unfortunately, *it was not understood as a symptom that TCP and IP were interdependent and therefore splitting it into two layers of the same scope was not a good idea* [36].
When Tinker Air Force Base joined the ARPANET in 1972, they wanted connections to two IMPs for redundancy. In theory that would have been a good idea. However, in the architecture of the ARPANET and Internet, that would not be possible [9]. The routing algorithm is not able to know that there are two physical wires going to the same place. Back in the day, and still to this day *IP[v6] addresses of all types are assigned to interfaces, not nodes* [29].

Because nodes in a network are identified by the name of an interface, it becomes hard to make advantage of other connections that a multihomed node has. When applications were to have their own address space, this problem could have been solved. In 1982, Jerry Saltzer published a paper on the naming and binding of network destinations [52]. His view on how naming should be used in networking is shown on the right side of Figure 2.2. In this scheme the node itself and the applications running at that node also have their own address for identification. This view on networking is not the reality in the current Internet, since there are no dedicated names or addresses for a node itself and the application(s) running on the node. The left side of Figure 2.2 shows the current situation in the Internet.
As mentioned before, prior to 1983, the ARPANET was running NCP. When TCP/IP was introduced, it ran on top of NCP. During that time, IP was an Internetwork Layer-protocol, while NCP was a Network Layer-protocol. When NCP was shut down in 1983 and replaced by TCP/IP, TCP/IP took over the network role and the internetwork layer was lost [11], as shown in Figure 2.1b. IP did not become a true Internetwork Protocol, but just a network protocol. The lack of having this internetwork transport layer, further complicates the multihoming and mobility problem. When NCP and TCP/IP were still running together, the IP address was not even the identifier for a host, it simply identifies the point-of-attachment of a host. The host itself was not addressed. *That would be the proper role of an "Internetwork" layer* [18].

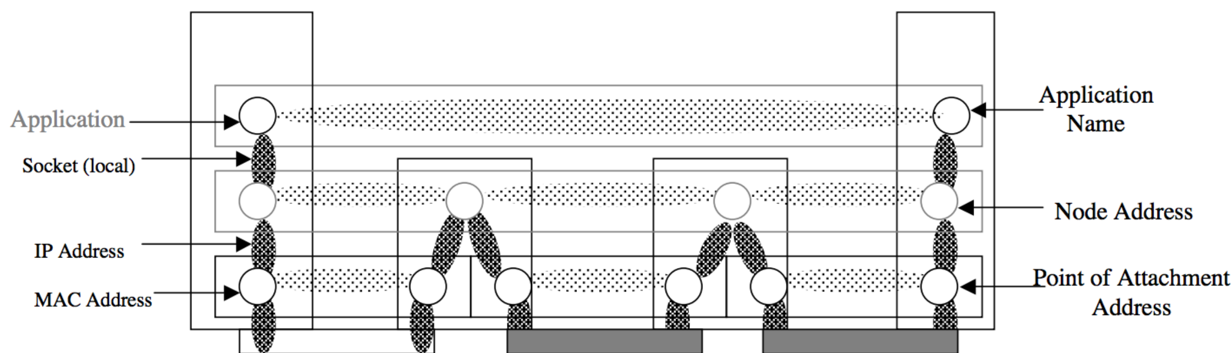


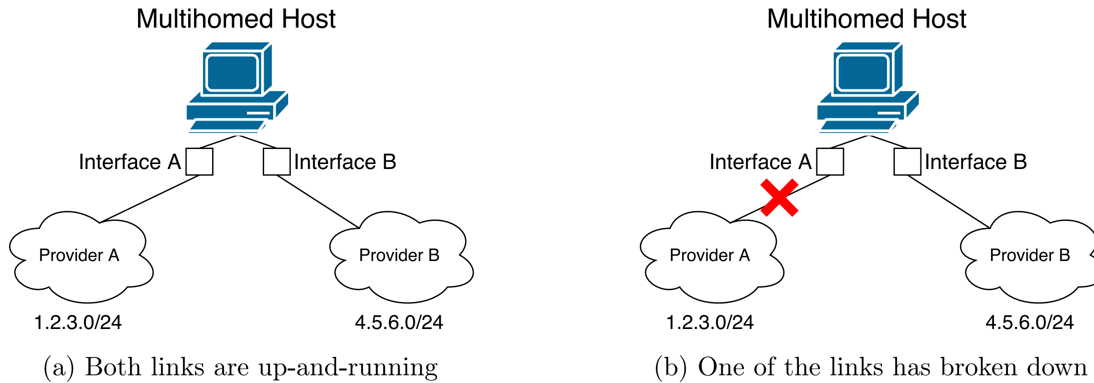
Figure 2.2: Saltzer's Naming Model (From: [10])

> "As a result, the Internet ceased to be an Internet and became basically a concatenation of IP networks with an end-to-end transport layer on top of it."
>
> (From: J. Day [11])

Instead of having multiple networks that were exchanging traffic, basically one big network was created (the Internet) that is interconnected. A big consequence of this decision is the fact that today's routing system is very complex, since both inter-domain and intra-domain routing is happening at the network layer [43].

2.2 Definition of Multihoming

Multihoming is the practise of connecting a host or a computer network to more than one network. This can be done in order to create reliability or improve performance, or to reduce cost [60] [26]. In a typical situation, a host is connected to simply one network. In some cases, it might be very useful to connect a host to multiple networks. This is called *Host Multihoming*. Improving the performance can be achieved by traffic engineering, it might be more efficient to route through another network. One can decrease the cost by choosing cheaper routes - it is possibly cost effective to route through another network [60]. An example of such a network topology is depicted in Figure 2.3. As shown in Figure 2.3b, when one of the interfaces fails it should rely on the other connected link. Without extra mechanisms to point traffic to the right direction, this is impossible. When an interface's point-of-attachment changes, there is a chance that the address changes, and there is a chance that the network prefix changes as well. This causes the network path to the node to change. Running sessions will break when the IP address changes, since it is established using the original IP address. From an Internet point-of-view, the address of the other interface could be located somewhere on the other side of the world. Another way of multihoming is multihoming per network, called *Site Multihoming* [56] [57], where a network is connected to multiple providers, while using its own range of addresses. This is later described in more detail.
As mentioned in the introduction of this report, mobility is closely related to multihoming. Mobility is the practise of *changing a single unit's point of attachment to the Internet and therefore its reachability in the Internet topology, while keeping its communication context active* [14] [26]. Tania Tronco states in her book that mobility is in a way very fast multihoming [56]. In case of mobility, the



(a) Both links are up-and-running (b) One of the links has broken down

Figure 2.3: Host Multihoming Example

device which is moving around does not necessarily need to be connected to multiple networks at the same time, but moves around and will be connected to networks other than its own home network. In today's Internet one would need extra protocols to make this work, e.g. LISP (Locator/Identifier Separation Protocol) [15], Mobile IPv4 [46], and Mobile IPv6 [47]. The current Internet solutions for multihoming will be further described in the next section.

2.3 Current solutions to the multihoming problem

The following subsections will describe the common solutions used today for doing multihoming. This includes multihoming in IPv4 networks, IPv6-only networks, and other architectures for using multihoming in a network, e.g. LISP.

2.3.1 Multihoming with IPv4

Classically, multihoming is done by connecting a network to multiple providers. Usually, this is realised by using a Provider-Independent address space (PI address space) range. PI is a block of IP addresses, which is assigned by a Regional Internet Registry (RIR) [51] to an organisation. An owner of this kind of address space needs to have a contract with an ISP to obtain routing for this block of addresses in order to make it accessible from the Internet. An advantage of having PI address space is the ability to change service providers without having to renumber the network. In order to perform multihoming, one needs to allocate a Provider-Independent address space prefix, which is reannounced by some or all of a network's peers. PI address space cannot be aggregated by an ISP and therefore it needs to be announced globally as is. This way of multihoming could provide fault-tolerance, improve the throughput, and reduce the network costs. However, the routing of the network needs to be carefully engineered. Multihoming works reasonably well in the network core, but that does not apply at the edge [6]. One of the biggest concerns with multihoming is the fact that, even when it is restricted to the large networks of the core, it will cause uncontrolled growth of the *default-free routing table*, because efficient route aggregation is generally not possible.
Multihoming is generally impossible to implement when using a single Provider-Aggregatable address space (PA address space) prefix. PA addresses are assigned by a single ISP. The route or routes covering those PA addresses is announced or propagated by one or more additional transit

providers[5] [1] [30]. The transit provider, which assigned the PA addresses, originates a set of routes which cover the site route set. The primary transit provider often originates or propagates the site route set as well as the covering aggregates. For more information, one can refer to RFC 4116 [1].
Since announcing the same PA prefix to multiple ISPs is not always possible, one solution would be to announce multiple PA prefixes, one per provider [6]. Using this approach, every host needs to have multiple addresses assigned, one per provider. However, having multiple addresses would require extra mechanisms for choosing a suitable source and destination address for each packet, and to properly route each outgoing packet to its destination [6].
If a network is connected to two different providers, a packet with a source address in the address range allocated to provider A, will usually not be accepted by provider B [16]. Provider B will treat this packet as a packet with a spoofed source address and will discard it [16]. Additionally, the prefix of provider A's network will not be re-announced by provider B. Therefore, destinations in provider A's prefix will not be reachable over the link of provider B.
The decision-making for choosing the source and destination addresses is typically done by the application layer. Once a connection has been established, it is no longer possible to change the source and destination address without breaking the (e.g.) TCP session. When one of the links of a multihomed host fails, the node will no longer be reachable on that particular address and is only available via the other interface with another address. This causes the node to be unreachable, since other nodes have been interacting with the node on the original chosen address. This makes the extra interface/address useless in the sense of redundancy. Another approach would be to use a transport protocol, which is capable of handling multiple addresses. One such protocol is Multipath TCP (MPTCP) [4]. MPTCP is an extension of TCP which is able to multiplex a single application layer flow over multiple Network Layer sub-flows, while trying to use as many distinct routes as possible and either tries to carry it over the most efficient route or tries to perform load balancing [6].
Even though multihoming with multiple addresses has been implemented for IPv4, it is generally not used. Host implementations are not able to handle the multiple address per interface that well [61].

2.3.2 Multihoming with IPv6

In IPv6 it is possible to do both multihoming with and without multiple addresses. RIPE NCC, Europe's RIR, has a special allocated IPv6 Provider-Independent address space [40]. Multihoming using PI address space works roughly the same as with IPv4, supporting traffic balancing across multiple providers, and maintaining existing TCP and UDP sessions through cut-overs. Multihoming with multiple addresses has been implemented for IPv6 [39]. For outgoing traffic, IPv6 needs to rely on support on the host with protocols like MPTCP and SCTP [55] or IPv6-specific protocols like SHIM6 [5].
Since IPv6 addresses are 96 bits longer than IPv4 addresses, there would be extra data that needs to be stored inside the routing tables. Some people have been saying that with IPv4 alone the increased size of the routing tables needed to handle multihoming will overwhelm current router hardware [7]. Using IPv6 will only increase the size of the routing table, since the Internet is not IPv6-only, but often dual-stack. Other people will argue that new hardware will be able to handle the increasing size of the routing tables due to cheaper memory [8].

[5] A 'transit provider' operates a network that directly provides connectivity to the Internet to one or more external sites. The connectivity provided extends beyond the transit provider's own site and its own direct customer networks. A transit provider's site is directly connected to the sites for which it provides transit.

Another solution for the multihoming problem[6] is Mobile IP (there is Mobile IP and Mobile IPv6). Mobile IP allows a mobile hosts to seamlessly move from its home domain to a foreign location without losing connectivity. Mobile IP provides two basic mechanisms for respectively discovery and registration. The discovery mechanism allows a node to detect its new point-of-attachment and the registration mechanism allows a node to update its location to a foreign agent[7], which will report to the home agent the mobile node's current location [45] [2].

2.3.3 Other solutions for the multihoming problem

Besides the implementations for multihoming mentioned above, there are also other solutions that try to mitigate the problems of multihoming in the Internet, e.g. the location problems and the routing problems. One of those solutions is the use of the Host Identity Protocol (HIP), therefore making this a host-based solution. HIP tries to separates the identifier and the locator of a node in the network. HIP introduces a Host Identity (HI) name space based on a Public Key Infrastructure [44]. When HIP is implemented within a network, one will eliminate the occurrences of IP addresses in applications and replace them with a (cryptographic) host identifier. This results in decoupling the Transport Layer from the Network Layer in TCP/IP.

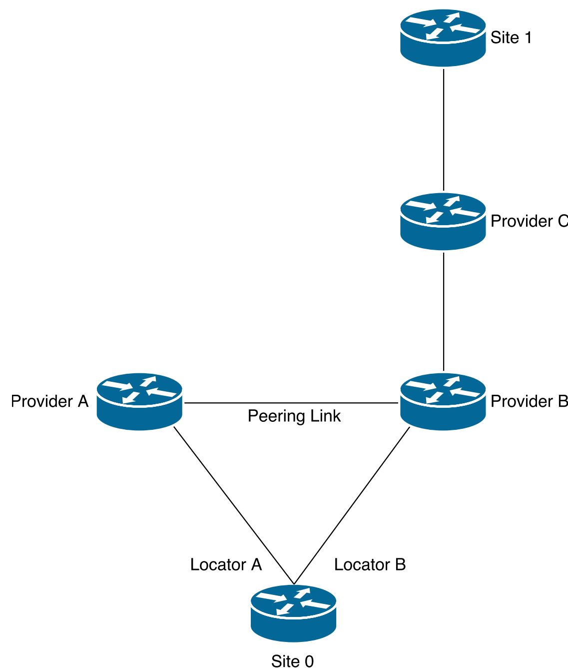


Figure 2.4: Reachability Failure (From: [42])

Another solution is to use the Locator/ID Separation Protocol (LISP). Unlike HIP, LISP is a network-based solution. LISP is a *map-and-encapsulate* protocol. LISP separates address space in identifiers for source and destination hosts, and routing locators where border routers act as routing locators for end systems. The way LISP does it is slightly different than with HIP. In LISP only one number space is used, namely the IP addresses[8] [15]. This can be both IPv4 and IPv6. When using LISP, one does not have to announce the locator of the node, only the endpoint identifier (EID). When using the map/encapsulating capabilities of LISP, it introduces the need for path discovery. Because the identifier and locator are separated in LISP, it is getting harder to determine if a particular destination locator is reachable. This general problem is also called *the Locator Path Liveness Problem*. This problem can be stated as follows:

[6] It focuses more on the mobility problem, than on the multihoming problem

[7] It is possible that a foreign agent is not yet deployed. Then, each mobile node in a foreign network is assigned its own co-located care-of-address

[8] Although, also arbitrary elements can be used, e.g. MAC addresses

> Given a set of source locators and a set of destination locators, can bi-directional connectivity be determined between <source locator,destination locator> address pairs?
> (From: D. Meyer, et al. [42])

The RFC describes the following example to explain this problem. Suppose a small topology, as shown in Figure 2.4.
In this scenario, Site 0 is multihomed to Provider A and Provider B. Site 0 has a PA locator from Provider A (Locator A), and a PA locator from Provider B (Locator B). In this case, Site 0 might 'advertise' that its EID prefixes can be reached through nodes Locator A and Locator B to its correspondent sites.
Suppose that a correspondent site, Site 1, is connected to Provider C, and that Site 0 has told Site 1 that it can reach Site 0 on either Locator A or Locator B. Also suppose that Site 1 chooses Locator A to reach Site 0, so that packets sourced from Site 1 destined for Site 0 traverse the path Site 1 → C → B → A → Site 0. If connectivity between Provider B and Provider A is disrupted, Locator A will not be reachable from Site 1. In this case, Site 1, must detect that Locator A is no longer reachable and use Locator B to restore connectivity [42].
The Locator Path Liveness Problem is exhibited in host-based architectures like SHIM6 and HIP, and in network-based architectures like LISP. This problem arises in subtly different ways, depending on the contents of the mapping database, e.g. EIDs, Resource Locators (RLOCs), or a combination of these two), and how knowledge is distributed between hosts and routing elements. Research in the past has proved that path discovery does not scale and therefore, solutions like HIP and LISP would not scale either [42].

As shown in this chapter, there are quite some problems with multihoming in the current Internet. In the current Internet model, the interface address names both the node itself and the interface (path) to that node. A lot of research has been performed on solving the multihoming problem, without actually solving the underlying problem. Most of the work performed is trying to create workarounds and it is debatable if these solutions will actually work and scale. These workarounds introduce more complexity, decrease efficiency, and make implementation and maintenance more difficult. Mainly, all the different ways of doing multihoming do not actually achieve the main advantage of having multihoming, namely resilience. It turns out to be very hard to make use of one's backup connection(s). In the next chapter, the Recursive InterNetwork Architecture will be introduced as one of the possible solutions to many Internet problems, including the multihoming problem.

Chapter 3

The Recursive InterNetwork Architecture

As mentioned in the introduction of this report, *the Recursive InterNetwork Architecture (RINA) is trying to work out the general principle in computer networking that applies to everything. RINA is a programmable networking approach, based on the Inter-Process Communication (IPC) paradigm, which will support high scalability, multihoming, built-in security, seamless access to real-time information, and operation in dynamic environments* [50].
In the current Internet we became all familiar with the principle of layering of the OSI model and the TCP/IP layered architecture. In these models, a layer is providing a service to the layer above it. An example would be the transport layer, which provides a virtual end-to-end channel to the application layer.
Since RINA is a very complicated, though interesting subject, it is not possible to describe the entire workings of RINA in one single paper. This chapter will give a short introduction to RINA. If one is interested in this subject, one can refer to John Day's book *Patterns in Network Architecture* [9], articles provided by the Pouzin Society [48], and e.g. the paper *Networking is IPC: A guiding principle to a better Internet* by John Day, Ibrahim Matta, and Karim Mattar [13].

3.1 Background Information

> "Networking provides the means by which processes on separate computer systems communicate, generalising the model of local inter-process communications."
>
> (From IRATI [23])

In RINA, the principle behind this new architecture is that networking is IPC and nothing else [13]. This basically unifies networking and distributed computing, since the network is a distributed application that provides IPC [24]. Another principle is that layers are recursive. The same protocol can be used repeatedly in a protocol stack, encapsulating each layer in another instance of itself. This implies that there is no need for special protocols per each layer and there is no fixed number of layers in the stack, like we have in TCP/IP [18]. In the current Internet, one is bound to the specific layer protocols, e.g. TCP or UDP for the transport layer. In RINA, there are basically only two protocols, an application protocol and a data transport protocol, respectively the Common Distributed Application Protocol (CDAP) and the Error and Flow Control Protocol (EFCP). The

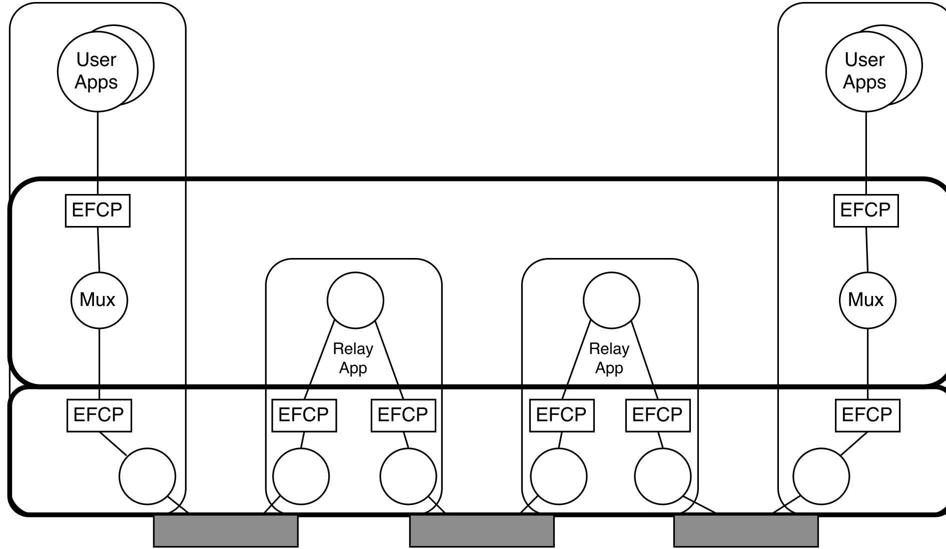


Figure 3.1: The IPC model (From [12])

CDAP protocol allows application processes to exchange structured data between each other[9]. The EFCP protocol provides the IPC connection and flows associated with each allocation request, providing the transfer of data between nodes [9]. Basically, one can have as many layers as needed. In the current Internet architecture one is limited to the fixed layering of TCP/IP. A Computer Science representation of the IPC principle is displayed in Figure 3.1. The two middle nodes could be compared to today's routers and the nodes at the left and right could be compared to today's servers. In this particular case there are two, so called, Distributed IPC Facilities (DIFs) and on top of the highest-level DIF the user applications are located. The grey bars on the bottom side of the diagram represent the physical (or virtual) connection between the different nodes.
In general, protocols are *not* bound to specific layers. One can use one or two protocols in one single layer. The implementation of such a scheme is simpler to implement than the TCP/IP stack, since there is no need for having separate protocols for, e.g. the data link layer and the network layer [18]. Each DIF is basically a repetition of the same protocols and functions. In RINA, a Distributed IPC Facility (DIF) can be seen as what one generally would refer to as a *layer*.

> "A Distributed IPC Facility (DIF) is an organising structure, grouping together application processes that provide IPC services and are configured under the same policies." (From IRATI [23])

As mentioned before, networking is in this case not a layered set of different functions, but rather a single layer of distributed IPC that repeats over different scopes. Such a scope could for instance be providing the same (e.g.) routing policy, but every scope can have other policies for, e.g. authorisation of access.

[9] Just like in an IP network, it is possible to do multicasting in RINA for, e.g. videostreaming

In order to interconnect different IPC processes (IPCPs), one needs to connect an IPCP to an existing DIF. This DIF can dynamically be created using a DIF discovery mechanism, or by creating a new DIF, when there is no suitable one available. This process of connecting IPCPs and DIFs is called *enrolling*. Enrolling IPCPs ensures that information about the IPCP is added, maintained, distributed, or deleted within a DIF. This information can be in the form of addressing, access-control, or other kind of policies to create instances and characterise a communication. After enrolling the IPCPs, the IPCPs are able to discover each other and become neighbours in the DIF [59]. When an application wants to exchange traffic, a flow is allocated within the IPCP. During this allocation phase, information to support data transfers is created, maintained, or deleted for a particular IPCP. An example of this kind of information would be the binding of an IPCP to a lower DIF (N-1). The allocation phase is done by the *Flow Allocator*. The Flow Allocator is responsible for creating, managing, and eventually deleting a flow [54]. Flows are stored in the flow table within the IPCP. When an application is finished exchanging traffic, the flow will be removed from the flow table.

3.2 RINA and Multihoming

Before an application can communicate with another application in the network, it needs to request service from the underlying DIF. This underlying DIF maps the destination application name to a node address. As mentioned before, a DIF can recursively provide transport services between source and destination application processes, using services of lower-level DIFs [35].

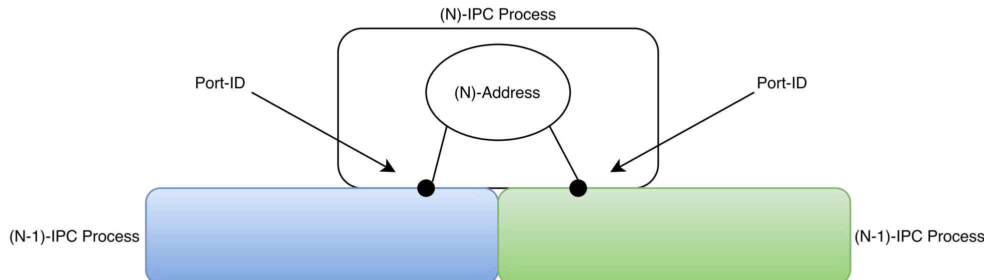


Figure 3.2: IPC Model with Multihoming

The route to the destination node address, the address to which the destination process is connected, is computed as a sequence of intermediate node addresses. At each routing hop, the next-hop node address is in turn recursively mapped to a lower-level node address by the underlying DIF. That lower-level node address can be viewed as the point-of-attachment of the higher-level node. Therefore, addresses within the RINA architecture are relative. A node address at a DIF level *N* is considered a node name by a lower-level *N-1* DIF. The directory service of the *N-1*-DIF needs to map this name to a node *N-1*-address. In the end, the node address maps to a specific path. Because of this way of binding to a specific path it makes it easier for RINA to deal with mobility and multihoming. If for any reason the active path to a node fails, RINA will map the node address of the destination node to another operational path. According to a paper by Matta et al. [35] the cost of such an operation would be very low, since the update is only local to the routing hop and the destination node address is mapped to a lower-level node address that resides within the operational

lower-level DIF. As mentioned in the previous chapter, in the Internet, the interface address of a node names both the node itself and the interface (path) to that node. This complicates the ability of managing mobility (and also multihoming). Figure 3.2 shows a simple RINA representation of an application that has been registered at two separate lower-level DIFs at the same time, creating a multihomed node. When a packet reaches a point in the path where it needs to make the decision which path to take, it forwards the packet based on the current underlying DIF leading to the destination process.

Basically, multihoming does not need a special mechanism in RINA, because it is a normal operation by RINA's design. When a node would connect to a network of another provider, the interface's address would change. However, the name of the host itself does not change, only the name of the interface(s) change. RINA's internal system would only need to change the mapping of the interface name, not the name of the node itself. John Day claims in his presentation at TCN 15 that a RINA network uses 50% up to 75% fewer addresses, and forwarding tables would proportionally be smaller as well compared to the current Internet [12].

3.3 Related RINA Projects

3.3.1 IRATI

RINA itself is only a network architecture and does not provide an implementation on its own. There are several projects that have been trying to create a working implementation of RINA. One of those implementations is Investigating RINA as an Alternative to TCP/IP (IRATI).

> "IRATI will advance the state of the art of RINA towards an architecture reference model and specification that are closer to enable implementations deployable in production scenarios. The design and implementation of a RINA prototype on top of Ethernet will permit the experiments and evaluation of RINA in comparison to TCP/IP."
>
> (From [34])

The IRATI project [34] was initially a Framework Programme 7 (FP7) project [17] funded by the European Union. As mentioned above, IRATI's goal is to achieve further exploration of this new Internet architecture. The main objectives of IRATI were to make enhancements to the RINA architecture reference model and specification, focussing on DIFs over Ethernet. Another objective was to create an actual open source prototype over Ethernet for UNIX-like operating systems. By creating a special DIF that is capable to run over Ethernet, it is easier to test RINA in an existing environment. Besides the DIF over Ethernet, IRATI also supports a DIF over TCP and UDP to run RINA over a current TCP/IP or UDP/IP network.

3.3.2 PRISTINE

Another closely related project to IRATI is PRISTINE [50]. This project is also funded by the European Union. PRISTINE implements RINA and creates programmable functions for congestion control, providing protection/resilience, facilitating more efficient topological routing and multi-layer management for handling configuration, performance, and security [49]. The work of this project group is a continuation of the work of the IRATI project. Features built by PRISTINE are often forked into the IRATI stack.

3.3.3 ARCFIRE

One of the newest projects, besides PRISTINE, is the ARCFIRE project [3]. Like IRATI and PRISTINE, this project is also funded by the European Union, but this time under the H2020 program [27]. The main objective for the ARCFIRE project is to demonstrate the large scale benefits of RINA, leveraging former European investments in Future Internet Testbeds (FIRE+) and in the development of the basic RINA technology. Their goal is to improve the IRATI software suite to make it possible to make large-scale experimental deployments with up to 100 nodes, supporting tens of hundreds of DIFs, running experiments for up to a week. The ARCFIRE project will closely work with the sister projects PRISTINE and IRATI.

Chapter 4

Experiments

This chapter describes the experiments that were conducted during this research project. First, the test environment will be described, followed by a description of the test scenarios and their results.

4.1 Test Environment

To perform tests with the IRATI stack, a virtual test environment was set up. KVM was used as hypervisor for running the test virtual machines. During the tests two different systems were used - a server with KVM virtual machines running a compiled IRATI stack, and a server running the IRATI Demonstrator tool. The IRATI Demonstrator tool is a command-line tool to easily test the IRATI stack in multi-node scenarios using small prefabricated VMs that include the IRATI stack. An advantage of this tool is that one does not need to compile and install the IRATI stack by hand. Another advantage is that configuration files are automatically generated using a topology configuration file. There are a couple of examples available to see how the topology configuration files work in order to create your own configuration file. All the nodes of this scenario are running in lightweight virtual machines[10] on top of a KVM hypervisor.

The characteristics of both servers and software versions, used during the experiments, are displayed in Table 4.1 and Table 4.2, respectively.

Hardware	*Description*
Device	Dell PowerEdge R210
CPU	Intel Xeon L3426 (4 cores, each 2 threads)
RAM	8GB (4x 2GB 1066 MHz)

Table 4.1: Hardware of the test servers

[10] The disk image is roughly 30 MB

Software	*Description*
Hosting OS	Ubuntu 16.04 LTS
Guest OS	Debian 8.5
GitHub Commit of IRATI Demonstrator [32]	1df754a
GitHub Commit of IRATI stack [33]	master (babd68b) pristine-1.5 (06c59b0)
Virtualisation software	libvirt-bin (1.3.1-1) qemu-kvm (1:2.5)
IRATI Wireshark [31]	c52c0e2

Table 4.2: Software of the test servers

The different VMs used during the experiments had the following characteristics, shown in Table 4.3.

Virtual hardware	*Description*
CPU	2 virtual CPUs
RAM	512MB
HDD	30GB[11]

Table 4.3: Information about VMs

4.2 Basic tests with IRATI

Just like Klomp & van Leur [37] did during their research, the compiled stack was tested by following the first two tutorials [19] [20] on the GitHub wiki of IRATI to get familiar with the workings of IRATI. These tutorials go through the steps of creating the appropriate configuration files and how to enroll the DIFs in the network.
After successfully testing the basic tutorials for simple connectivity, the third tutorial [21] on IRATI's wiki was followed, which contained multiple spanning DIFs to resemble multiple providers. The expectations were that it would just simply work by following the instructions of the tutorial. Unfortunately, there were some problems with the description of the tutorial. In the meantime, the version of the stack was updated on GitHub and the way of writing configuration files changed considerably. Therefore, enrolling the DIFs failed during the setup. The topology of this tutorial is shown in Figure 4.1.
This topology resembles a provider network being connected to a customer network over different DIFs. An application, like *rina-echo-time*[12], will register to the *multi-provider.DIF* DIF to get connectivity from the customer network to the provider network. One of the characteristics of this

[11] The disk size only applies to the VMs used for the compiled IRATI stack, the VMs used in combination with the IRATI Demonstrator are using a disk image of 30MB
[12] The rina-echo-time is a ping application suitable for RINA networks

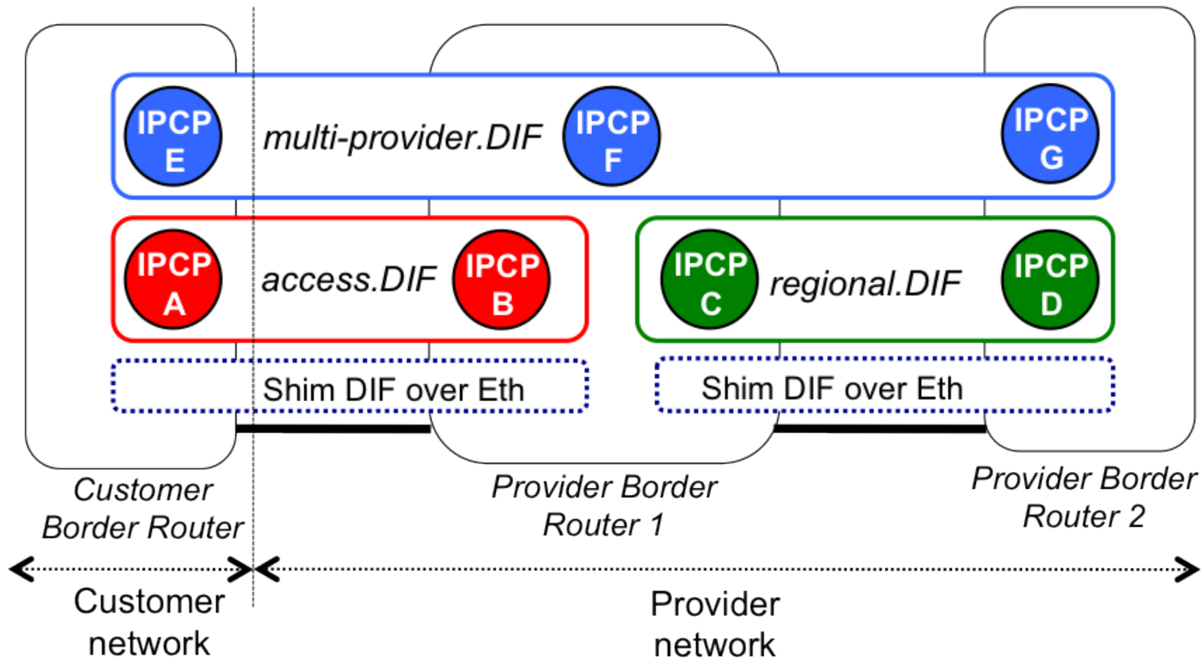


Figure 4.1: RINA Topology of Tutorial 3 (From: [21])

tutorial was to connect the *access.DIF* DIF in a secure way using a password or using keys. Due to lack of time and the outdated wiki pages on the GitHub, no extra time was spent to make the scenario of the tutorial work. A GitHub issue was filed to notify the developers of the outdated wiki pages.

4.3 Multihoming with IRATI

To test if the IRATI stack is capable of handling multihoming according to the RINA design [25], the following setup was used:

- One node that represents a multihomed host, connecting to two different providers, named *Provider A* and *Provider B*;
- One node per provider with two different network connections (DIFs), namely an access DIF and a regional DIF;
- One node that represents an upstream provider to which both providers are connected;
- One node that represents a server somewhere on "the Internet".

In order to test the multihoming capabilities of the IRATI stack, the topology shown in Figure 4.2 was used. All nodes of the topology are connected using a virtual bridge and use different VLANs to communicate to each other. The VLANs are represented in the Figure as *rbr*<VLAN id>. To represent a multi-provider network, multiple DIFs are used. Node 1 is connected to two

different providers. The multihomed node is connected to two different provider's access networks by means of an access DIF. The provider nodes represent a router within a provider's network connecting the access DIF to a provider's regional DIF. Finally, the regional DIFs are connected to a shared upstream provider. Node 5 is located "somewhere in the Internet" and connects through the upstream DIF to Node 1 over one of the two overlaying DIFs.

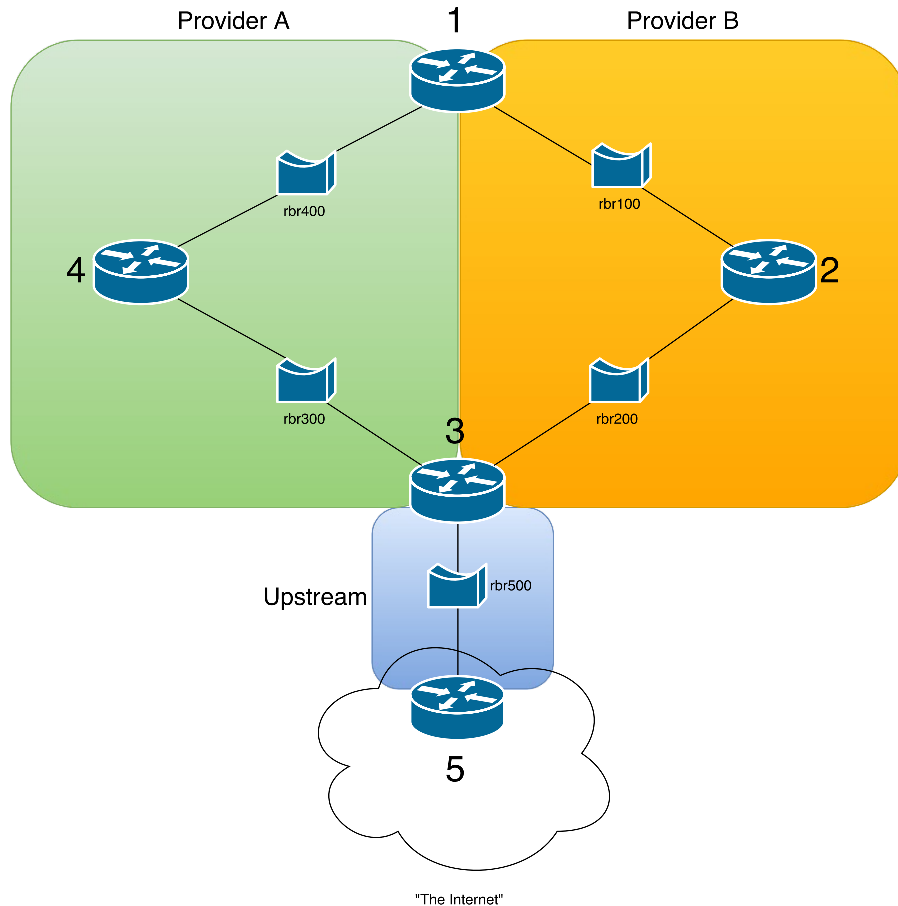


Figure 4.2: Network Topology

4.3.1 Test cases

During the experiments, two main tests were conducted. One test to see if it is possible to create a multi-DIF topology that includes a multihomed node and two different providers. The other test looks at the possibility of using the multihoming capabilities of a RINA network with an available application. Unfortunately, not a lot of applications are available today that are capable to run on a RINA network. One of the working programs is the before-mentioned *rina-echo-time* application, which is part of the *rina-tools*. The *rina-echo-time* application resembles the same features as the well-known ICMP echo request, also known as *ping*. The *rina-echo-time* application needs to be run as a server and as a separate client.

The expected result of these experiments was that the multi-DIF topology and the multihoming capabilities of IRATI will work. The main reason for this was that the theoretical architecture supports both features out of the box. Pinging a server from a multihomed node should be working. However, since this is still an experimental implementation, there might be some problems with the code of the implementation itself.

To easily test the topology shown in Figure 4.2, the IRATI Demonstrator was used to automatically create the appropriate configuration files for the IPCPs and the different DIFs in the topology. By doing this, one can easily shutdown the small VMs by running the *down.sh* script of the Demonstrator tool and bring them back up with the *up.sh* script. The scripts for generating and bringing up and down the VMs is explained in more detail in Appendix B. Using the IRATI Demonstrator saves a considerable amount of time, since one does not need to compile the IRATI stack and one does not need to configure everything by hand.

Because the traffic flowing between nodes in a RINA network is not 'normal' TCP/IP traffic, the traffic would not be readable in a packet capture file when using Wireshark. IRATI developed a special version of Wireshark to be able to get the data units of a RINA network readable. Unfortunately, since Klomp & van Leur's [37] research in January, no extra work has been done by IRATI to maintain the Wireshark repository. The problems they experienced during their research do still exist. Therefore, the IRATI Wireshark was not used during the research.

4.3.2 IRATI Demonstrator

To create a topology with the IRATI Demonstrator, one needs to create a topology configuration file. The configuration file specifies which virtual shim interfaces need to be made and which DIFs need to be created. These virtual interfaces will connect to an automatically generated virtual bridge to make connections between the VMs. The first DIF in this topology is the shim DIF, which makes the connection between the network interface and the IRATI stack. One needs to specify the DIF or DIFs one wants to span over the shim interfaces. In the topology of this experiment, multiple DIFs span over each other. First, the Access DIFs, Regional DIFs, and the Upstream DIF are spanned over the shim interfaces. And finally, the end-to-end DIF that connects Node 1 to Node 5 in the topology is spanned over the Access, Regional, and Upstream DIF of the appropriate providers. A small excerpt of the IRATI Demonstrator configuration file is shown below in Listing 4.1.

Listing 4.1: Excerpt of IRATI Demonstrator configuration

```
# Defining Shim interfaces
# 400 is a shim-eth-vlan DIF, with nodes 1 and 4
eth 400 0Mbps 1 4
# 300 is a shim-eth-vlan DIF, with nodes 4 and 3
eth 300 0Mbps 3 4
# 500 is a shim-eth-vlan DIF, with nodes 3 and 5
eth 500 0Mbps 3 5

# Defining DIFs
dif providerAAccess 1 400
dif providerAAccess 4 400

dif providerARegional 4 300
dif providerARegional 3 300

dif upstream 3 500
```

```
dif upstream 5 500

dif providerAInternet 1 providerAAccess
dif providerAInternet 4 providerAAccess providerARegional
dif providerAInternet 3 providerARegional upstream
dif providerAInternet 5 upstream
```

Because the DIFs are not visible in network topologies like in Figure 4.2, the IRATI developers use a specific way of representing the stacking of DIFs in a network. Such a representation of the experiments, carried out during this project, are shown in Figure 4.3 and Figure 4.4. Since two providers in one single representation makes the diagram harder to read and understand, the two different providers are split over two different diagrams.

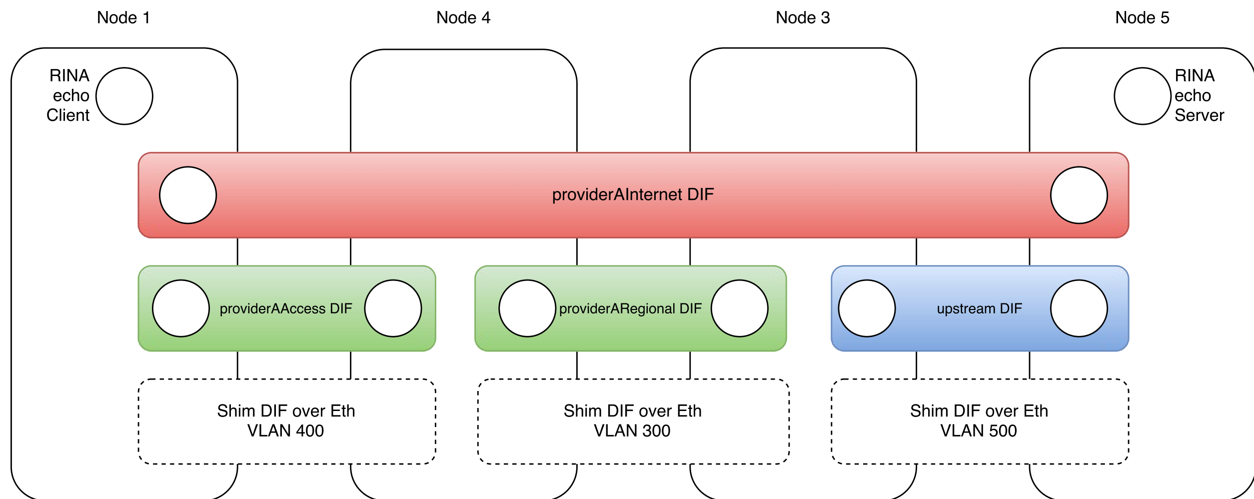


Figure 4.3: RINA representation of Provider A

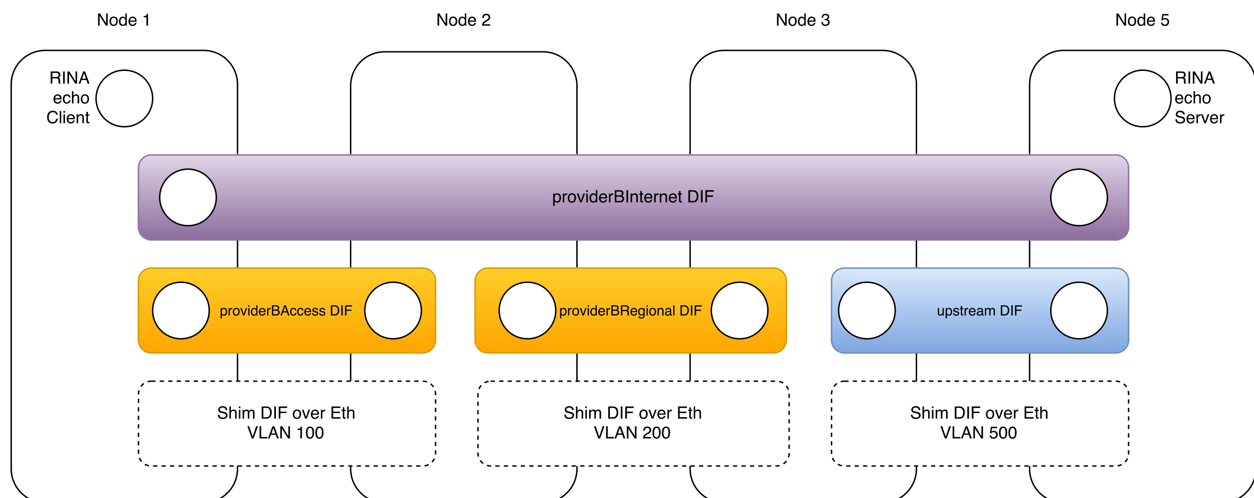


Figure 4.4: RINA representation of Provider B

In the RINA representation, the different DIFs are clearly visible. The bottom DIFs represent the shim DIFs, which make the connection between Ethernet and the RINA stack. On top of the shim DIF, the access, regional, or upstream DIFs are spanned. And finally, on top of the these DIFs, the end-to-end DIF is spanned (*providerXInternet DIF*).

After running the generation script, one can start the VMs using the IRATI Demonstrator *start.sh* script. A more detailed description of how the configurations are generated is described in Listing B.2 in Appendix B. A *rina-echo-time* server was started on Node 5 and an attempt was made to ping the server using the *rina-echo-time* client on Node 1. The execution of the commands for running the client and server are shown in Listings 4.2 and 4.3.

Listing 4.2: *rina-echo-time* client

```
root@stockholm:/home/koen/demonstrator# ./access.sh 1
Accessing buildroot VM 1
Warning: Permanently added '[localhost]:2223' (ECDSA) to the list of known hosts.
# rina-echo-time -c 4
2088(1468416549)#librina.logs (DBG): New log level: INFO
2088(1468416549)#librina.nl-manager (INFO): Netlink socket connected to local port 2088
Flow allocation time = 4.9679 ms
SDU size = 20, seq = 0, RTT = 1.8312 ms
SDU size = 20, seq = 1, RTT = 1.9319 ms
SDU size = 20, seq = 2, RTT = 1.9906 ms
SDU size = 20, seq = 3, RTT = 2.0144 ms
SDUs sent: 4; SDUs received: 4; 0% SDU loss
Minimum RTT: 1.8312 ms; Maximum RTT: 2.0144 ms; Average RTT:1.942 ms; Standard deviation:
    0.081594 ms
```

Listing 4.3: *rina-echo-time* server

```
root@stockholm:/home/koen/demonstrator# ./access.sh 5
Accessing buildroot VM 5
Warning: Permanently added '[localhost]:2227' (ECDSA) to the list of known hosts.
# rina-echo-time -l
1087(1468416540)#librina.logs (DBG): New log level: INFO
1087(1468416540)#librina.nl-manager (INFO): Netlink socket connected to local port 1087
1087(1468416548)#rina-echo-app (INFO): New flow allocated [port-id = 4]
1087(1468416551)#rina-echo-app (INFO): Flow torn down remotely [port-id = 4]
```

As shown above, the pinging between the nodes was successful. However, further investigation showed that the data for the *rina-echo-time* server was going over one and the same path all the time. It turned out that the RINA implementation does not dynamically select a path, but the mapping between the applications (*rina-echo-time*) and the DIFs (either internetProviderA.DIF or internetProviderB.DIF) is done statically. This is visible when looking at the Routing Information Base (RIB) of the IPCP[13], as shown in Listing 4.4. When the *rina-echo-time* client is started, a flow is created for the *rina.apps.echotime.client:1::* to *rina.apps.echotime.server:1::*. As shown in the Listing, an underlying flow is called which uses the *providerBInternet* DIF. The other DIF, the *providerAInternet* DIF, is not mentioned in the flow table.

[13] The RIB is queried using the IPCP console. The way of connecting to the console is described in Appendix D.

Listing 4.4: IPCP RIB of VM1

```
Name: /fa/flows/key=17-5; Class: Flow; Instance: 48
Value: * State: 2
* Is this IPC Process the requestor of the flow? 1
* Max create flow retries: 1
* Hop count: 3
* Source AP Naming Info: rina.apps.echotime.client:1::
* Source address: 17
* Source port id: 5
* Destination AP Naming Info: rina.apps.echotime.server:1::* Destination addres: 21
* Destination port id: 4
* Connection ids of the connection supporting this flow: +
Src CEP-id 0; Dest CEP-id 0; Qos-id 1
* Index of the current active connection for this flow: 0

[...]

Name: /ipcManagement/irm/underflows/portId=4; Class: UnderlayingFlow; Instance: 32
Value: Local app name: providerBInternet.1.IPCP-1--Remote app name: providerBInternet.2.IPCP
    -1--
N-1 DIF name: providerBAccess.DIF; port-id: 4
Flow characteristics: Jitter: 0; Delay: 0
In order delivery: 0; Partial delivery allowed: 1
Max allowed gap between SDUs: -1; Undetected bit error rate: 0
Average bandwidth (bytes/s): 0; Average SDU bandwidth (bytes/s): 0
Peak bandwidth duration (ms): 0; Peak SDU bandwidth duration (ms): 0
```

The mapping between applications and DIFs is done using a special configuration file called *da.map*. When using the IRATI Demonstrator, this file is automatically generated. When compiling the IRATI stack, one needs to create this file. An example of this file is shown in Listing 4.5.

Listing 4.5: Example configuration file: da.map

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

As shown in Listing 4.5, the *rina-echo-time* server and client are mapped to one specific DIF, namely *providerBInternet.DIF*. This phenomenon is further investigated in the second experiment, described in the next section.

Another interesting result of using the IRATI Demonstrator was the constant crashing of the VMs. It appeared that the memory consumption of the VMs kept increasing. After running the *rina-echo-time* application for a while, to see if there were any lost packets during the test, the memory consumption of the VM increased linearly. When the memory is almost completely used, the whole VM crashes and needs to be restarted. This increasing memory consumption also happened when the *rina-echo-time* server and client were not running. Using the script described in Appendix E, the free memory was logged every second until the VM crashed. The graph plot of the log file is shown in Figure 4.5.

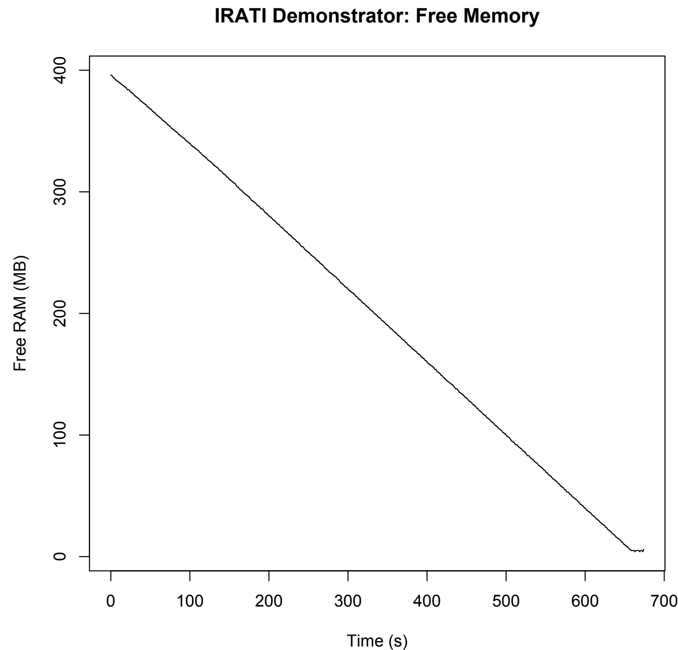


Figure 4.5: Free RAM using IRATI Demonstrator (RAM: 512 MB)

One of the possibilities for this increase of memory usage over time is the debugging option of the IRATI stack. This option is enabled by default when compiling the IRATI stack. Since the IRATI stack is already compiled when using the IRATI Demonstrator, one cannot be entirely sure if the debugging is enabled. Klomp & van Leur [37] had problems with the debugging option enabled, because the debugging was filling up the disks. This was not the case in this experiment, only RAM was affected. When generating the VMs using the IRATI Demonstrator, one can choose the size of the RAM of every VM. When the size of the RAM was increased to 1024MB, the pattern of free memory was still linear and went down until the VM crashed. It was unclear which specific version of the IRATI stack was used for the IRATI Demonstrator. In the next part of the experiments, it was tested if the compiled IRATI stack showed the same memory behaviour. Since the configuration files generated by the IRATI Demonstrator are in principle the same, these configuration files were reused during the next experiment. By using these configuration files, a lot of time was saved when configuring the IRATI stack in the next experiment. It was assumed that the internal workings of the IRATI Demonstrator and the compiled IRATI stack were the same.

4.3.3 IRATI stack

To further investigate the possibilities of multihoming with the IRATI stack, and to see if the compiled stack also has problems with running out of memory, new VMs were created. For this experiment, the same topology is used as during the IRATI Demonstrator test. In order to run the IRATI stack, one needs to compile and install the special kernel with RINA support and the

required libraries and RINA tools. When compiling the IRATI stack, the debugging option was disabled, because Klomp & van Leur [37] had problems with filling disks when the debugging option was enabled. A detailed description of which options were used for compiling the IRATI stack is described in Appendix A. During the IRATI Demonstrator test, a couple of configuration files were generated for the IPC manager and the different DIFs. To see if the memory problem also appeared in the VMs with the compiled IRATI stack, the memory test mentioned in the previous section was executed on these VMs as well. As shown in Figure 4.6, the VMs with the compiled IRATI stack did not show this behaviour and the memory consumption was stable throughout the whole experiment. This means that there is probably something wrong with the used disk image of the IRATI Demonstrator VMs, since the increasing memory usage is not seen in the compiled stack VMs. Apparently, the IRATI Demonstrator and the compiled IRATI stack are not entirely the same.

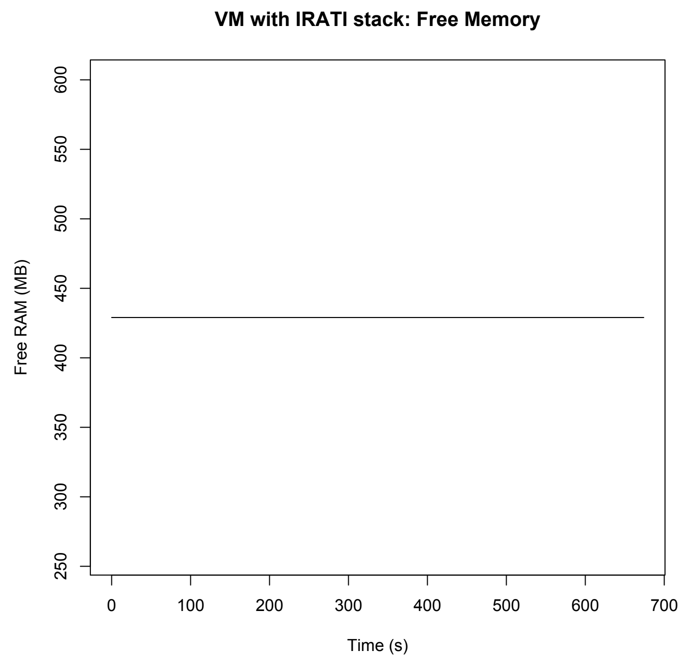


Figure 4.6: Free RAM using VM with compiled IRATI stack (RAM: 512MB)

As shown in the IRATI Demonstrator experiment, the mapping of the application and the DIFs is done statically. This forces traffic to a specific DIF, instead of dynamically choosing a path from application to application. First, an attempt was made to use the DIF allocation file to register one application to multiple DIFs at the same time. This caused the stack to crash. This is probably due to the fact that the IRATI libraries are not able to process multiple DIF assignments for one application at the same time, when using the *da.map* file. A second attempt was made by issuing the *-d* flag of the *rina-echo-time* application, to specify to which DIF the *rina-echo-time* application needs to be registered. According to IRATI developer Eduard Grasa, the implementation of IRATI supports multiple registrations, but the IRATI applications do not exploit this capability yet.

The developers agreed to add the function that allows for registering multiple DIFs to the *rina-tools*, and specially the *rina-echo-time* application. Using this extra capability, it is now possible to actually test if registering an application to multiple DIFs works. Initially, it was only possible to either map the applications to a specific DIF using the *da.map* file or by explicitly stating at which DIF one wants to register an application (e.g. *rina-echo-time -l -d specificdif.DIF*). The developers[14] released the fix through the *pristine-1.5* branch under commit 9e82180[15]. The kernel and user parts needed to be recompiled using the *pristine-1.5* stack, therefore, new VMs were created to be sure that the newest kernel was installed. Another test was performed with the patched *RINA tools* to see if the IRATI implementation would now be able to connect to a multihomed server. With the change in the *rina-tools*, it is possible to use the *-d* flag with multiple DIFs, separated by a comma, as shown in Listing 4.8.

Listing 4.6: *rina-echo-time* client 1

```
root@multi1:/home/koen# rina-echo-time -c 4 -d providerAInternet.DIF
6902(1467986643)#librina.logs (DBG): New log level: INFO
6902(1467986643)#librina.nl-manager (INFO): Netlink socket connected to local port 6902
Flow allocation time = 38.375 ms
SDU size = 20, seq = 0, RTT = 5.2941 ms
SDU size = 20, seq = 1, RTT = 16.758 ms
SDU size = 20, seq = 2, RTT = 16.083 ms
SDU size = 20, seq = 3, RTT = 1.568 ms
SDUs sent: 4; SDUs received: 4; 0% SDU loss
Minimum RTT: 1.568 ms; Maximum RTT: 16.758 ms; Average RTT:9.9258 ms; Standard deviation:
    7.6571 ms
```

Listing 4.7: *rina-echo-time* client 2

```
root@multi1:/home/koen# rina-echo-time -c 4 -d providerBInternet.DIF
7167(1467986656)#librina.logs (DBG): New log level: INFO
7167(1467986656)#librina.nl-manager (INFO): Netlink socket connected to local port 7167
Flow allocation time = 62.392 ms
SDU size = 20, seq = 0, RTT = 54.72 ms
SDU size = 20, seq = 1, RTT = 1.6436 ms
SDU size = 20, seq = 2, RTT = 34.567 ms
SDU size = 20, seq = 3, RTT = 10.775 ms
SDUs sent: 4; SDUs received: 4; 0% SDU loss
Minimum RTT: 1.6436 ms; Maximum RTT: 54.72 ms; Average RTT:25.426 ms; Standard deviation:
    23.958 ms
```

Listing 4.8: *rina-echo-time* server

```
root@multi5:/home/koen# rina-echo-time -l -d providerAInternet.DIF,providerBInternet.DIF
5131(1467986629)#librina.logs (DBG): New log level: INFO
5131(1467986629)#librina.nl-manager (INFO): Netlink socket connected to local port 5131
5131(1467986629)#rina-echo-time (INFO): Application registered in DIF providerAInternet.DIF
5131(1467986629)#rina-echo-time (INFO): Application registered in DIF providerBInternet.DIF
5131(1467986643)#rina-echo-app (INFO): New flow allocated [port-id = 4]
5131(1467986646)#rina-echo-app (INFO): Flow torn down remotely [port-id = 4]
5131(1467986656)#rina-echo-app (INFO): New flow allocated [port-id = 5]
5131(1467986659)#rina-echo-app (INFO): Flow torn down remotely [port-id = 5]
```

[14] Special thanks to Eduard Grasa

[15] https://github.com/IRATI/stack/commit/9e82180ecf728eb8ce64891c9a6067d3916d4868

As shown in Listing 4.8, the *rina-echo-time* application is registered to two separate DIFs, *providerAInternet.DIF* and *providerBInternet.DIF*. The two DIFs use each a different port-id to distinguish the different connections to the node and application. After spinning up the server side, one can connect to the client, either via *providerAInternet.DIF* or *providerBInternet.DIF*. As shown in Listings 4.6 and 4.7, a connection is made using the *rina-echo-time* client over a specified DIF. When the client is starting, the server will detect a new connection request and will create a new flow for this traffic. When the application is finished sending traffic, the flow will be removed from the flow table. These flows are shown in Listing 4.9. These flows are from the RIB on VM5 in the topology.

Listing 4.9: IPCP RIB of VM5

```
Name: /fa/flows/key=17-5; Class: Flow; Instance: 51
Value: * State: 2
* Is this IPC Process the requestor of the flow? 0
* Max create flow retries: 1
* Hop count: 3
* Source AP Naming Info: rina.apps.echotime.client:1::
* Source address: 17
* Source port id: 5
* Destination AP Naming Info: rina.apps.echotime.server:1::* Destination addres: 21
* Destination port id: 4
* Connection ids of the connection supporting this flow: +
Src CEP-id 0; Dest CEP-id 0; Qos-id 1
* Index of the current active connection for this flow: 0

Name: /ipcManagement/irm/underflows/portId=2; Class: UnderlayingFlow; Instance: 32
Value: Local app name: providerAInternet.5.IPCP-1--Remote app name: providerAInternet.3.IPCP
    -1--
N-1 DIF name: upstream.DIF; port-id: 2
Flow characteristics: Jitter: 0; Delay: 0
In order delivery: 0; Partial delivery allowed: 1
Max allowed gap between SDUs: -1; Undetected bit error rate: 0
Average bandwidth (bytes/s): 0; Average SDU bandwidth (bytes/s): 0
Peak bandwidth duration (ms): 0; Peak SDU bandwidth duration (ms): 0

[...]

Name: /fa/flows/key=17-6; Class: Flow; Instance: 57
Value: * State: 2
* Is this IPC Process the requestor of the flow? 0
* Max create flow retries: 1
* Hop count: 3
* Source AP Naming Info: rina.apps.echotime.client:1::
* Source address: 17
* Source port id: 6
* Destination AP Naming Info: rina.apps.echotime.server:1::* Destination addres: 21
* Destination port id: 5
* Connection ids of the connection supporting this flow: +
Src CEP-id 0; Dest CEP-id 0; Qos-id 1
* Index of the current active connection for this flow: 0

Name: /ipcManagement/irm/underflows/portId=3; Class: UnderlayingFlow; Instance: 32
Value: Local app name: providerBInternet.5.IPCP-1--Remote app name: providerBInternet.3.IPCP
    -1--
N-1 DIF name: upstream.DIF; port-id: 3
Flow characteristics: Jitter: 0; Delay: 0
In order delivery: 1; Partial delivery allowed: 1
Max allowed gap between SDUs: -1; Undetected bit error rate: 0
Average bandwidth (bytes/s): 0; Average SDU bandwidth (bytes/s): 0
Peak bandwidth duration (ms): 0; Peak SDU bandwidth duration (ms): 0
```

Because the application is registered to a node, it does not matter to which DIF it is connected to, or to which DIFs it is connected. At this point, it is not possible to simply run a *rina-echo-time* client without specifying through which DIF it should connect. It would be nice when one does not have to specify over which connection the client should connect. However, it might not be preferable if a client randomly connects to any available DIF. In the future, this might be part of (e.g.) the access policies one can have inside a DIF.

As shown in this chapter, the IRATI Demonstrator is a very handy tool to quickly test topologies and quickly test connectivity within the network. For actual tests the compiled IRATI stack is more preferable, because the compiled stack did not run out of memory during the experiments and turned out to be more stable for tests. As proven by the experiments, the IRATI stack does support the multihoming capabilities, but the applications do not fully exploit this feature yet. Due to the patch of the *rina-echo-time* application, one can test the multi-registration feature, albeit fairly limited. In the next chapter, the results of the experiments will be discussed.

Chapter 5

Discussion

This chapter will discuss the results of the experiments described in the previous chapter.

5.1 IRATI Demonstrator

The IRATI Demonstrator turned out to be a very useful tool to quickly generate a small RINA test network. As shown in Chapter 4, the IRATI Demonstrator seems to suffer from a memory problem. The VMs will keep consuming RAM in a linear fashion until there is not enough RAM available to keep everything up and running. Basically, the whole VM, including the IPC processes will stop functioning. Since the IRATI Demonstrator should be used for quick testing, this is not a very big problem. During the experiment, it simply did what it should do: showing that the test topology would work in a compiled IRATI environment. For future researchers it would be nice if the IRATI Demonstrator did not show the behaviour constantly increasing the memory consumption.
As shown in the experiments, the IRATI Demonstrator was able to create a multi-DIF topology. The DIFs were enrolled correctly and, in principle, the connectivity between applications over multiple nodes could be tested. However, to get a better view on performance, the compiled stack would be more realistic, because it has more resources to do the processing of information.

5.2 Compiled IRATI stack

In order to compile the IRATI stack, one needs a lot of patience, since there are quite some required packages needed before one can compile the IRATI stack. Compiling the kernel and all the user space parts (libraries, RINA-tools, etc.) take the most time of all. It takes easily up to three hours to finish the compiling and installation of the IRATI stack (without configuring it). It would save a lot of time if the IRATI kernel, libraries and applications are available through the package repository.

Unfortunately, a lot of documentation seems to be outdated. Since Klomp & van Leur's research in January 2016, a lot of improvements have been made to the code of the IRATI stack. Unfortunately, some vital parts of the wiki are not updated. It would be really helpful for future research if the documentation of IRATI would be updated. The outdated documentation gave some problems when trying to go through the tutorials listed on the wiki. An example would be the way of communicating with the IPC console. It turned out that in the meantime, the way of interacting with the

IPC console changed from using a Telnet connection, to a socket file. Another thing that changed is the configuration syntax. Some tutorials are using an older version of the IRATI stack, and therefore, at least one of the tutorials does not work anymore. During this project, there was not enough time to try and fix the tutorial using the new configuration syntax. For future researchers it would be very convenient if the configuration files for the tutorials are updated to the newest syntax.

Klomp & van Leur describe in their research that the IRATI Wireshark, which is available on the IRATI GitHub page, was not able to clearly view the traffic flows due to the inability of the Wireshark dissectors to automatically adjust to the IRATI configurations. During the experiments, the IRATI Wireshark was tried to see if it is working now. Unfortunately, there has not been any activity in the IRATI Wireshark repository since the 6th of March 2015. When writing this report, it is therefore still not entirely possible to easily view the traffic flows, without the need of manually changing the Wireshark dissectors.

One of the main goals of this project was to see if multihoming is possible using the current IRATI stack. At the moment, it seems that the multihoming capabilities of RINA are not fully integrated yet into the IRATI stack. Since the mapping between applications and DIFs is done statically at this point in time, one cannot entirely speak of multihoming. For now, one still needs to specify to which DIF one wants to connect. By request, one of the developers of the IRATI stack, Eduard Grasa, made a patch for the *RINA-echo-time* application, so it is capable of registering the server to multiple DIFs simultaneously. The RINA tools in the *pristine-1.5* branch are updated to include this patch. As shown in the experiments, it is now possible to connect to a multihomed *RINA-echo-time* server. However, the RINA tools are still incapable of automatically selecting which DIF to use when connecting to, for instance, a *RINA-echo-time* server. According to the developers, the H2020 ARCFIRE project is going to research and develop a dynamic DIF allocator to create this feature for the IRATI stack. Meanwhile, the PRISTINE project will focus on developing and improving the routing capabilities of the IRATI stack to make the routing for multihoming and mobility as efficient as possible.

Throughout the report, mobility is also mentioned a couple of times, but not really tested during the experiments. During an email conversation with one of the developers, it became clear that at this moment, there has been no tests with IRATI and mobility. Especially the H2020 ARCFIRE project is going to research and test the mobility capabilities of IRATI in the near future. For this moment, they will focus on WiFi only, since there are no other technologies for mobility supported yet.

Chapter 6

Conclusion

The current Internet faces quite some challenges and one of these challenges is the multihoming problem. Throughout the years a lot of effort has been put in solutions for multihoming, but none of them actually tries to solve the root of the problem.
The Recursive InterNetwork Architecture (RINA) is a new computer network architecture where networking is seen as Inter-Process Communication (IPC). The architecture tries to solve many of the current Internet's problems, including multihoming. By design RINA supports multihoming, mobility, security, and much more. Where in the current Internet only the interfaces are named, RINA also names the nodes and the applications. This would solve the multihoming problem in theory, since the node and the application are identifiable by their own address and not both by the interface name.
One of the major implementation projects is the IRATI project. They are working on implementing the RINA architecture and making enhancements to the specification. As shown by the experiment executed during this research, it is not possible to use the multihoming capabilities of RINA using the current IRATI stack version. At this time the mapping between applications and DIFs is done statically. Therefore, one cannot speak of multihoming entirely, since dynamically assigning applications to DIFs is not yet implemented. However, it is now possible to register the *RINA-echo-time* server to multiple DIFs at the same time to facilitate the multihoming capabilities. Therefore, RINA is capable of doing multihoming, however, the current IRATI applications are not fully capable of exploiting the multihoming capabilities yet.

Chapter 7

Future Work

When using the IRATI Demonstrator, the VMs seem to have a memory problem. The memory consumption goes linearly up, until the VMs completely crash. Since the compiled IRATI stack does not seem to experience this behaviour, it is likely that there is something wrong with the disk image used by the IRATI Demonstrator. It is not a very big problem, since it should be used for quick topology tests. However, it would be worth it for the developers to fix this, probably, small problem.

Currently, the DIF allocation is still static. However, the dynamic DIF allocator will be developed somewhere in the coming two years. It would be very interesting to see, if the topology used in this research project would work without the need of manually assigning DIFs to an application.

During the experiments, mobility has not been one of the priorities to look into. In the coming two years, the H2020 ARCFIRE project is going to research and develop features for the IRATI stack, concerning mobility. When they would finish their project, it would be interesting to see if this would scale in a realistic topology.

Chapter 8

Bibliography

[1] Abley, J., Lindqvist, K., Davies, E., Black, B., and Gill, V. IPv4 Multihoming Practices and Limitations. RFC 4116, RFC Editor, July 2005. http://www.rfc-editor.org/rfc/rfc4116.txt.

[2] Åhlund, C., and Zaslavsky, A. Multihoming with Mobile IP. In *IEEE International Conference on High Speed Networks and Multimedia Communications* (2003), Springer, pp. 235–243.

[3] ARCFIRE. ARCFIRE: Objectives. http://ict-arcfire.eu/index.php/about-arcfire/objectives/, 2016. [Online].

[4] Barré, S., Paasch, C., and Bonaventure, O. Multipath TCP: from theory to practice. In *NETWORKING 2011*. Springer, 2011, pp. 444–457.

[5] Barré, S., Ronan, J., and Bonaventure, O. Implementation and evaluation of the Shim6 protocol in the Linux kernel. *Computer Communications* (2011). http://dx.doi.org/10.1016/j.comcom.2011.03.005.

[6] Boutier, M., and Chroboczek, J. Source-specific routing. In *IFIP Networking Conference (IFIP Networking), 2015* (2015), IEEE, pp. 1–9.

[7] Castro Casales, A., Germán Duarte, M., Yannuzzi, M., and Masip Bruin, X. Insights on the Internet routing scalability issues. In *1st Workshop on Multilayer Networks* (2009).

[8] Committee on the Internet in the Evolving Information Infrastructure. *The Internet's Coming of Age.* National Academy of Sciences, 2001.

[9] Day, J. *Patterns in network architecture: a return to fundamentals.* Pearson Education, 2007.

[10] Day, J. Things They Never Taught You About Naming and Addressing. http://rina.tssg.org/docs/FutureNetTutorialPart2-100415.pdf, May 2010.

[11] Day, J. How in the Heck do you lose a layer!? In *Network of the Future (NOF), 2011 International Conference on the* (2011), IEEE, pp. 135–143.

[12] Day, J. Connected Communities Need Solid Foundations. https://tnc15.terena.org/core/presentation/173, June 2015. TERENA TNC15.

[13] Day, J., Matta, I., and Mattar, K. Networking is IPC: A Guiding Principle to a Better Internet. In *Proceedings of the 2008 ACM CoNEXT Conference* (New York, NY, USA, 2008), CoNEXT '08, ACM, pp. 67:1–67:6.

[14] Ernst, T. Network Mobility Support Goals and Requirements. RFC 4886 (Informational), July 2007.

[15] Farinacci, D., Fuller, V., Meyer, D., and Lewis, D. The Locator/ID Separation Protocol (LISP). RFC 6830, RFC Editor, January 2013. http://www.rfc-editor.org/rfc/rfc6830.txt.

[16] Ferguson, P., and Senie, D. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. BCP 38, RFC Editor, May 2000. http://www.rfc-editor.org/rfc/rfc2827.txt.

[17] FP7. Research & Innovation: Framework Programme 7. https://ec.europa.eu/research/fp7. [Online].

[18] Goldstein, F., and Day, J. Moving beyond TCP/IP. *Pouzin Society,(April 2010), http://rina. tssg. org/docs/PSOC-MovingBeyondTCP. pdf* (2010).

[19] Grasa, E. Tutorial 1: DIF over a VLAN (point to point DIF). https://github.com/IRATI/stack/wiki/Tutorial-1:-DIF-over-a-VLAN-(point-to-point-DIF). [Online; accessed 09-June-2016].

[20] Grasa, E. Tutorial 2: DIF over two VLANS. https://github.com/IRATI/stack/wiki/Tutorial-2:-DIF-over-two-VLANS. [Online; accessed 09-June-2016].

[21] Grasa, E. Tutorial 3: Security provider net. https://github.com/IRATI/stack/wiki/Tutorial-3:-Security-provider-net. [Online; accessed 09-June-2016].

[22] Grasa, E. Getting Started: How to install and build IRATI. https://github.com/IRATI/stack/wiki/Getting-Started, 2016. [Online].

[23] Grasa, E. Introduction to RINA. http://irati.eu/introduction-to-rina/, 2016. [Online; accessed 20-June-2016].

[24] Grasa, E., Lichtner, O., Rysavy, O., Asgari, H., Day, J., and Chitkushev, L. From Protecting protocols to layers: designing, implementing and experimenting with security policies in RINA. ICC 2016, Kuala Lumpur, May 2016.

[25] Grasa, E., Trouva, E., Phelan, P., de Leon, M. P., Day, J., Matta, I., Chitkushev, L. T., and Bunch, S. Design principles of the Recursive InterNetwork Architecture (RINA). http://www.future-internet.eu/fileadmin/documents/fiarch23may2011/06-Grasa_DesignPrinciplesOTheRecursiveInterNetworkArchitecture.pdf. [Online].

[26] Gundu, N. Mobility vs Multihoming. In *Seminar on Internetworking* (2004).

[27] H2020. Horizon 2020 Projects. http://horizon2020projects.com/. [Online].

[28] Hafner, K., and Lyon, M. *Where wizards stay up late: The origins of the Internet.* Simon and Schuster, 1998.

[29] Hinden, R., and Deering, S. IP Version 6 Addressing Architecture. RFC 4291, RFC Editor, February 2006. http://www.rfc-editor.org/rfc/rfc4291.txt.

[30] Huston, G. Architectural approaches to multi-homing for ipv6. RFC 4177, RFC Editor, September 2005. http://www.rfc-editor.org/rfc/rfc4177.txt.

[31] IRATI. GitHub - IRATI Wireshark. https://github.com/IRATI/wireshark, 2015. [Online].

[32] IRATI. GitHub - IRATI Demonstrator. https://github.com/IRATI/demonstrator, 2016. [Online].

[33] IRATI. GitHub - IRATI stack. https://github.com/IRATI/stack/, 2016. [Online].

[34] IRATI. Investigating RINA as an Alternative to TCP/IP. http://irati.eu, 2016. [Online; accessed 3-June-2016].

[35] Ishakian, V., Akinwumi, J., Esposito, F., and Matta, I. On supporting mobility and multihoming in recursive internet architectures. *Computer Communications 35*, 13 (2012), 1561–1573.

[36] Kent, C. A., and Mogul, J. C. Fragmentation Considered Harmful. *SIGCOMM Comput. Commun. Rev. 25*, 1 (Jan. 1995), 75–87.

[37] Klomp, J., and van Leur, J. An Assessment of the IRATI Implementation. Master's thesis, University of Amsterdam, 2016. http://rp.delaat.net/2015-2016/p22/report.pdf.

[38] Koymans, K. History of Unix and the Internet. https://www.os3.nl/_media/2015-2016/courses/cia/history.pdf, 2016. [Online; accessed 09-June-2016].

[39] Lamparter, D., and Smirnov, A. Destination/Source Routing. Internet-Draft draft-ietf-rtgwg-dst-src-routing-02, IETF Secretariat, May 2016. http://www.ietf.org/internet-drafts/draft-ietf-rtgwg-dst-src-routing-02.txt.

[40] Martinez, J. P. Provider Independent (PI) IPv6 Assignments for End User Organisations. Policy 2006-01, RIPE NCC, April 2006. https://www.ripe.net/participate/policies/proposals/2006-01.

[41] McKenzie, A. INWG and the Conception of the Internet: An Eyewitness Account. *IEEE Annals of the History of Computing 33*, 1 (2011), 66–71.

[42] Meyer, D., and Lewis, D. Architectural Implications of Locator/ID Separation. Internet-Draft draft-meyer-loc-id-implications-01, IETF Secretariat, January 2009. http://www.ietf.org/internet-drafts/draft-meyer-loc-id-implications-01.txt.

[43] Mills, D. Exterior Gateway Protocol formal specification. RFC 904, RFC Editor, April 1984.

[44] Moskowitz, R., and Nikander, P. Host Identity Protocol (HIP) Architecture. RFC 4423, RFC Editor, May 2006.

[45] Periklis, Z. A Study of RINA, a novel Internet Architecture. Master's thesis, Hochschule Darmstadt, 2015. http://www.islab.demokritos.gr/gr/html/MyThesis_Zisis.pdf.

[46] Perkins, C. IP Mobility Support for IPv4, Revised. RFC 5944, RFC Editor, November 2010. http://www.rfc-editor.org/rfc/rfc5944.txt.

[47] Perkins, C., Johnson, D., and Arkko, J. Mobility Support in IPv6. RFC 6275, RFC Editor, July 2011. http://www.rfc-editor.org/rfc/rfc6275.txt.

[48] Pouzin Society. RINA Education: Highlights. http://pouzinsociety.org/education/highlights. [Online].

[49] PRISTINE. PRISTINE main achievements and exploitation prospects. http://ict-pristine.eu/wp-content/uploads/2013/12/Pristine_project_FI_factsheet.pdf, 2015. [Online, accessed 10-June-2016].

[50] PRISTINE. PRISTINE - Overview. http://ict-pristine.eu/?page_id=48, 2016. [Online; accessed 10-June-2016].

[51] RIPE NCC. FAQ: ISPs. https://www.ripe.net/participate/member-support/info/faqs/isp-related-questions. [Online].

[52] Saltzer, J. H. On the Naming and Binding of Network Destinations. RFC 1498, RFC Editor, August 1993. http://www.rfc-editor.org/rfc/rfc1498.txt.

[53] SecureNet. History of Unix and the Internet. http://www.securenet.net/members/shartley/history/tcp_ip.htm, 2016. [Online; accessed 09-June-2016].

[54] Small, J. Patterns in Network Security: An analysis of architectural complexity in securing Recursive Inter-Network Architecture networks. Master's thesis, Boston University Metropolitan College, 2012. http://rina.tssg.org/docs/js-002.7-patterns_in_network_security.pdf.

[55] Stewart, R. Stream Control Transmission Protocol. RFC 4960, RFC Editor, September 2007. http://www.rfc-editor.org/rfc/rfc4960.txt.

[56] Tronco, T. *New Network Architectures: The Path to the Future Internet.* Springer, 2010.

[57] van Beijnum, I. A Look at Multihoming and BGP, 2002.

[58] Vrijders, S., Staessens, D., Colle, D., Salvestrini, F., Grasa, E., Tarzan, M., and Bergesio, L. Prototyping the recursive internet architecture: the IRATI project approach. *IEEE Network 28*, 2 (March 2014), 20–25.

[59] Wang, Y., Esposito, F., Matta, I., and Day, J. Resursive InterNetwork Architecture (RINA) - Boston University Prototype: Programming Manual (version 1.0). http://csr.bu.edu/rina/papers/BUCS-TR-2013-013.pdf, 2013. [Online].

[60] Wikipedia. Multihoming — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Multihoming&oldid=723278381, 2016. [Online; accessed 03-June-2016].

[61] Winter, R., Faath, M., and Ripke, A. Multipath TCP Support for Single-homed End-systems. Internet-Draft draft-wr-mptcp-single-homed-07, IETF Secretariat, March 2016. http://www.ietf.org/internet-drafts/draft-wr-mptcp-single-homed-07.txt.

Glossary

ARCFIRE Large-scale RINA benchmark on FIRE. 17, 32, 34

ARPA Advanced Research Projects Agency. 6

CDAP Common Distributed Application Protocol. 13, 14

DARPA Defence Advanced Research Projects Agency. 6

default-free routing table The default-free routing table, or default-free zone (DFZ) refers to the collection of all Internet Autonomous Systems (AS) that do not require a default route to route a packet to any destination. 9

DIF Distributed IPC Facility. iv, v, xiv, 1, 14–17, 19–25, 27–30, 32–34

EFCP Error and Flow Control Protocol. 13, 14

FIRE Future Internet Research and Experimentation. 17

HIP Host Identity Protocol. 11, 12

IMP Interface Message Processor, the first generation of gateways (routers) in the ARPANET. 7

IP Internet Protocol. 4–7, 11, 13, 14

IPC Inter-Process Communication. iv, v, xiv, xv, 4, 13, 14, 27, 31–33

IPCP Inter-Process Communication Process. 15, 22, 24

IPv4 Internet Protocol version 4. 9–11

IPv6 Internet Protocol version 6. 9–11

IRATI Investigating RINA as an Alternative To TCP/IP. i, iii, v, xiv, xvii, 1, 5, 16–20, 22–27, 31–34

ISP Internet Service Provider. 9, 10

LISP Locator/Identifier Separation Protocol. 9, 11, 12

MPTCP Multipath TCP. 10

NCP Network Control Program. 6, 7

PA address space Acronym for Provider-Aggregatable address space. 9, *see* Provider-Aggregatable address space

PI address space Acronym for Provider-Independent address space. 9, *see* Provider-Independent address space

PRISTINE Programmability In RINA for European Supremacy of Virtualised Networks. ii, 16, 17, 32

Provider-Aggregatable address space Provider-aggregatable (PA) address space is a block of IP addresses assigned by a Regional Internet Registry to an Internet Service Provider, which can be aggregated into a single route advertisement for improved Internet routing efficiency. 9

Provider-Independent address space A provider-independent address space (PI) is a block of IP addresses assigned by a Regional Internet Registry directly to an end-user organisation. 9

RIB Routing Information Base. 24

RINA Recursive InterNetwork Architecture. ii, 1, 4, 5, 13–17, 19–21, 24, 26, 27, 31–33

RIR Regional Internet Registry. 10

SCTP Stream Control Transmission Protocol. 10

SHIM6 Site Multihoming by IPv6 Intermediation. 10

TCP Transmission Control Protocol. 4–7, 10, 11, 13, 14

UDP User Datagram Protocol. 10

Appendices

Appendix A

Compiling and Installing the IRATI stack

In order to compile the IRATI stack, a special IRATI kernel needs to be compiled and a couple of applications and libraries need to be compiled and installed. A description of how the kernel and the user space packages need to be compiled and installed is described on the IRATI wiki [22].
The *Getting Started* page is a bit outdated (Debian 7.0 was used for the wiki, during the research Debian 8.5 was used). The following required packages were installed:

- kernel-package
- libncurses5-dev
- autoconf
- automake
- libtool
- pkg-config
- git
- g++
- openjdk-7-jdk
- maven
- libssl-dev
- protobuf-compiler
- libprotobuf-dev
- libnl-genl-3-dev
- libnl-3-dev
- libpcre3-dev

In the *Getting Started* page it is mentioned that one needs to add the testing repository. Since Debian 8 is used, one does not need to do this any longer.
Another required package needs to be build from source, namely *swig*. In this research, version 3.0.10 was used:

Listing A.1: Compiling and Installing Swig

```
koen@vm1:~$ http://prdownloads.sourceforge.net/swig/swig-3.0.10.tar.gz
koen@vm1:~$ tar -xzvf swig-3.0.10.tar.gz
koen@vm1:~$ cd swig-3.0.10
koen@vm1:~$ ./configure
koen@vm1:~$ make
koen@vm1:~$ sudo make install
```

When all the required packages are installed, the repository of the stack can be cloned:

Listing A.2: Cloning IRATI stack (master branch)

```
koen@vm1:~$ git clone git://github.com/IRATI/stack.git
```

During tests, it turned out that the master was behind the branch of PRISTINE, which is still actively maintained. Later results showed that the *pristine-1.5* branch was more stable. To clone that specific branch the following command needs to be run:

Listing A.3: Cloning IRATI stack (pristine-1.5 branch)

```
koen@vm1:~$ git clone -b pristine-1.5 git://github.com/IRATI/stack.git
```

Klomp & van Leur [37] found out that by default the debugging option is enabled, which results in the disks being filled up with debugging information. To avoid that, the debugging option needs to be disabled.

Listing A.4: Configuring the kernel options

```
root@vm1:/home/koen# cd stack/
root@multi1:/home/koen/stack#
root@multi1:/home/koen/stack# cd linux
root@multi1:/home/koen/stack/linux# make menuconfig
```

In the RINA menu, one needs to make sure to disable the debugging option and save the .config file afterwards before compiling the kernel.
Since everything is now ready for compilation and installation of the stack, one needs to run the following script, which will take care of the compiling and installing of the kernel and the user parts:

Listing A.5: Compiling and Installing the Kernel and User parts

```
root@vm1:/home/koen# cd stack/
root@multi1:/home/koen/stack#
root@multi1:/home/koen/stack# ./install-from-scratch
```

Appendix B

Configuration Experiment #1: IRATI Demonstrator

This appendix contains the configuration file for the IRATI Demonstrator (multihoming.conf) and how the IRATI Demonstrator is used:

Listing B.1: multihoming-topology.conf

```
# This config is for multihoming use case

# R1 ----- R2 ----- R3 ---- R5
# |                 |
# |                 |
# -------- R4 ----- |

# 100 is a shim-eth-vlan DIF, with nodes 1 and 2
eth 100 0Mbps 1 2

# 200 is a shim-eth-vlan DIF, with nodes 2 and 3
eth 200 0Mbps 2 3

# 400 is a shim-eth-vlan DIF, with nodes 1 and 4
eth 400 0Mbps 1 4

# 300 is a shim-eth-vlan DIF, with nodes 4 and 3
eth 300 0Mbps 3 4

# 500 is a shim-eth-vlan DIF, with nodes 3 and 5
eth 500 0Mbps 3 5

# DIFs
dif providerAAccess 1 400
dif providerAAccess 4 400

dif providerBAccess 1 100
dif providerBAccess 2 100

dif providerARegional 4 300
dif providerARegional 3 300

dif providerBRegional 2 200
dif providerBRegional 3 200

dif upstream 3 500
dif upstream 5 500
```

```
dif providerAInternet 1 providerAAccess
dif providerAInternet 4 providerAAccess providerARegional
dif providerAInternet 3 providerARegional upstream
dif providerAInternet 5 upstream

dif providerBInternet 1 providerBAccess
dif providerBInternet 2 providerBAccess providerBRegional
dif providerBInternet 3 providerBRegional upstream
dif providerBInternet 5 upstream
```

To create the needed configuration files for the IPC manager and the DIFs, one needs to run the following commands:

Listing B.2: Generating configuration files and starting VMs using IRATI Demonstrator

```
# Generating the configuration files
root@stockholm:/home/koen/demonstrator# ./gen.py -m 512 -c multi3.conf

# Generating the necessary virtual bridge interfaces and spinning up the VMs
root@stockholm:/home/koen/demonstrator# ./up.sh

# Accessing VMs
root@stockholm:/home/koen/demonstrator# ./access.sh <name_of_vm>

# Stopping the VMs and removing the virtual bridge interfaces
root@stockholm:/home/koen/demonstrator# ./down.sh
```

This is the configuration for the application mapping (da.map):

Listing B.3: da.map

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

Appendix C

Configuration Experiment #2: VMs with compiled IRATI stack

In this Appendix all the configurations used for the IRATI stack VMs are described.

C.1 Configuration for the IPC manager

The following Listings describe the configuration files for all the IPC managers for every separate VM. These files describe the configuration for the different DIFs and which IPC processes to create. The files are generated by the IRATI Demonstrator and are slightly changed to match the interfaces on the full-Debian VMs.

Listing C.1: ipcmanager.conf for Node 1

```
{
  "configFileVersion" : "1.4.1",
  "localConfiguration" : {
    "installationPath" : "/bin",
    "libraryPath" : "/lib",
    "logPath" : "/var/log",
    "consoleSocket" : "/var/run/ipcm-console.sock",
    "pluginsPaths" : ["/lib/rinad/ipcp"]
  },
    "difConfigurations": [
        {
            "name": "100",
            "template": "shimeth.1.100.dif"
        },
        {
            "name": "400",
            "template": "shimeth.1.400.dif"
        },
        {
            "name": "providerBAccess.DIF",
            "template": "normal.providerBAccess.dif"
        },
        {
            "name": "providerAAccess.DIF",
            "template": "normal.providerAAccess.dif"
        },
        {
            "name": "providerAInternet.DIF",
            "template": "normal.providerAInternet.dif"
```

```
        },
        {
            "name": "providerBInternet.DIF",
            "template": "normal.providerBInternet.dif"
        }
    ],
    "ipcProcessesToCreate": [
        {
            "apInstance": "1",
            "apName": "eth.1.IPCP",
            "difName": "100"
        },
        {
            "apInstance": "1",
            "apName": "eth.2.IPCP",
            "difName": "400"
        },
        {
            "apInstance": "1",
            "apName": "providerBAccess.1.IPCP",
            "difName": "providerBAccess.DIF",
            "difsToRegisterAt": [
                "100"
            ]
        },
        {
            "apInstance": "1",
            "apName": "providerAAccess.1.IPCP",
            "difName": "providerAAccess.DIF",
            "difsToRegisterAt": [
                "400"
            ]
        },
        {
            "apInstance": "1",
            "apName": "providerAInternet.1.IPCP",
            "difName": "providerAInternet.DIF",
            "difsToRegisterAt": [
                "providerAAccess.DIF"
            ]
        },
        {
            "apInstance": "1",
            "apName": "providerBInternet.1.IPCP",
            "difName": "providerBInternet.DIF",
            "difsToRegisterAt": [
                "providerBAccess.DIF"
            ]
        }
    ]
    }
}
```

Listing C.2: ipcmanager.conf for Node 2

```
{
  "configFileVersion" : "1.4.1",
  "localConfiguration" : {
    "installationPath" : "/bin",
    "libraryPath" : "/lib",
    "logPath" : "/var/log",
    "consoleSocket" : "/var/run/ipcm-console.sock",
    "pluginsPaths" : ["/lib/rinad/ipcp"]
  },
```

```
    "difConfigurations": [
        {
            "name": "100",
            "template": "shimeth.2.100.dif"
        },
        {
            "name": "200",
            "template": "shimeth.2.200.dif"
        },
        {
            "name": "providerBRegional.DIF",
            "template": "normal.providerBRegional.dif"
        },
        {
            "name": "providerBAccess.DIF",
            "template": "normal.providerBAccess.dif"
        },
        {
            "name": "providerBInternet.DIF",
            "template": "normal.providerBInternet.dif"
        }
    ],
    "ipcProcessesToCreate": [
        {
            "apInstance": "1",
            "apName": "eth.1.IPCP",
            "difName": "100"
        },
        {
            "apInstance": "1",
            "apName": "eth.2.IPCP",
            "difName": "200"
        },
        {
            "apInstance": "1",
            "apName": "providerBRegional.2.IPCP",
            "difName": "providerBRegional.DIF",
            "difsToRegisterAt": [
                "200"
            ]
        },
        {
            "apInstance": "1",
            "apName": "providerBAccess.2.IPCP",
            "difName": "providerBAccess.DIF",
            "difsToRegisterAt": [
                "100"
            ]
        },
        {
            "apInstance": "1",
            "apName": "providerBInternet.2.IPCP",
            "difName": "providerBInternet.DIF",
            "difsToRegisterAt": [
                "providerBAccess.DIF",
                "providerBRegional.DIF"
            ]
        }
    ]
}
```


Listing C.3: ipcmanager.conf for Node 3


```
{
```

```
"configFileVersion" : "1.4.1",
"localConfiguration" : {
  "installationPath" : "/bin",
  "libraryPath" : "/lib",
  "logPath" : "/var/log",
  "consoleSocket" : "/var/run/ipcm-console.sock",
  "pluginsPaths" : ["/lib/rinad/ipcp"]
},
  "difConfigurations": [
      {
          "name": "200",
          "template": "shimeth.3.200.dif"
      },
      {
          "name": "300",
          "template": "shimeth.3.300.dif"
      },
      {
          "name": "500",
          "template": "shimeth.3.500.dif"
      },
      {
          "name": "providerBRegional.DIF",
          "template": "normal.providerBRegional.dif"
      },
      {
          "name": "providerARegional.DIF",
          "template": "normal.providerARegional.dif"
      },
      {
          "name": "upstream.DIF",
          "template": "normal.upstream.dif"
      },
      {
          "name": "providerAInternet.DIF",
          "template": "normal.providerAInternet.dif"
      },
      {
          "name": "providerBInternet.DIF",
          "template": "normal.providerBInternet.dif"
      }
  ],
  "ipcProcessesToCreate": [
      {
          "apInstance": "1",
          "apName": "eth.1.IPCP",
          "difName": "200"
      },
      {
          "apInstance": "1",
          "apName": "eth.2.IPCP",
          "difName": "300"
      },
      {
          "apInstance": "1",
          "apName": "eth.3.IPCP",
          "difName": "500"
      },
      {
          "apInstance": "1",
          "apName": "providerBRegional.3.IPCP",
          "difName": "providerBRegional.DIF",
          "difsToRegisterAt": [
              "200"
          ]
```

```
        },
        {
            "apInstance": "1",
            "apName": "providerARegional.3.IPCP",
            "difName": "providerARegional.DIF",
            "difsToRegisterAt": [
                "300"
            ]
        },
        {
            "apInstance": "1",
            "apName": "upstream.3.IPCP",
            "difName": "upstream.DIF",
            "difsToRegisterAt": [
                "500"
            ]
        },
        {
            "apInstance": "1",
            "apName": "providerAInternet.3.IPCP",
            "difName": "providerAInternet.DIF",
            "difsToRegisterAt": [
                "providerARegional.DIF",
                "upstream.DIF"
            ]
        },
        {
            "apInstance": "1",
            "apName": "providerBInternet.3.IPCP",
            "difName": "providerBInternet.DIF",
            "difsToRegisterAt": [
                "providerBRegional.DIF",
                "upstream.DIF"
            ]
        }
    ]
}
```

Listing C.4: ipcmanager.conf for Node 4

```
{
  "configFileVersion" : "1.4.1",
  "localConfiguration" : {
    "installationPath" : "/bin",
    "libraryPath" : "/lib",
    "logPath" : "/var/log",
    "consoleSocket" : "/var/run/ipcm-console.sock",
    "pluginsPaths" : ["/lib/rinad/ipcp"]
  },
    "difConfigurations": [
        {
            "name": "300",
            "template": "shimeth.4.300.dif"
        },
        {
            "name": "400",
            "template": "shimeth.4.400.dif"
        },
        {
            "name": "providerARegional.DIF",
            "template": "normal.providerARegional.dif"
        },
        {
            "name": "providerAAccess.DIF",
```

```
            "template": "normal.providerAAccess.dif"
        },
        {
            "name": "providerAInternet.DIF",
            "template": "normal.providerAInternet.dif"
        }
    ],
    "ipcProcessesToCreate": [
        {
            "apInstance": "1",
            "apName": "eth.1.IPCP",
            "difName": "300"
        },
        {
            "apInstance": "1",
            "apName": "eth.2.IPCP",
            "difName": "400"
        },
        {
            "apInstance": "1",
            "apName": "providerARegional.4.IPCP",
            "difName": "providerARegional.DIF",
            "difsToRegisterAt": [
                "300"
            ]
        },
        {
            "apInstance": "1",
            "apName": "providerAAccess.4.IPCP",
            "difName": "providerAAccess.DIF",
            "difsToRegisterAt": [
                "400"
            ]
        },
        {
            "apInstance": "1",
            "apName": "providerAInternet.4.IPCP",
            "difName": "providerAInternet.DIF",
            "difsToRegisterAt": [
                "providerAAccess.DIF",
                "providerARegional.DIF"
            ]
        }
    ]
}
```

Listing C.5: ipcmanager.conf for Node 5

```
{
  "configFileVersion" : "1.4.1",
  "localConfiguration" : {
    "installationPath" : "/bin",
    "libraryPath" : "/lib",
    "logPath" : "/var/log",
    "consoleSocket" : "/var/run/ipcm-console.sock",
    "pluginsPaths" : ["/lib/rinad/ipcp"]
  },
    "difConfigurations": [
        {
            "name": "500",
            "template": "shimeth.5.500.dif"
        },
        {
            "name": "upstream.DIF",
```

```
            "template": "normal.upstream.dif"
        },
        {
            "name": "providerAInternet.DIF",
            "template": "normal.providerAInternet.dif"
        },
        {
            "name": "providerBInternet.DIF",
            "template": "normal.providerBInternet.dif"
        }
    ],
    "ipcProcessesToCreate": [
        {
            "apInstance": "1",
            "apName": "eth.1.IPCP",
            "difName": "500"
        },
        {
            "apInstance": "1",
            "apName": "upstream.5.IPCP",
            "difName": "upstream.DIF",
            "difsToRegisterAt": [
                "500"
            ]
        },
        {
            "apInstance": "1",
            "apName": "providerAInternet.5.IPCP",
            "difName": "providerAInternet.DIF",
            "difsToRegisterAt": [
                "upstream.DIF"
            ]
        },
        {
            "apInstance": "1",
            "apName": "providerBInternet.5.IPCP",
            "difName": "providerBInternet.DIF",
            "difsToRegisterAt": [
                "upstream.DIF"
            ]
        }
    ]
}
```

C.2 Other configuration files

When all the configuration files are included in the appendices of this report, the report will become rather long. Therefore, only the important configuration files are included in the appendices. These configuration files and all other configuration files are located at the following location: `https://github.com/koenveelenturf/irati-test-rp2`.

C.3 Setting up VLANs

In order to run the IRATI stack, a Bash script was created for every separate VM to set up the VLANs on the interfaces and to load the *shim-eth-vlan*, *rina-default-plugin*, and *normal-ipcp* kernel modules. These modules make sure that VLAN-over-Ethernet connections can communicate with the IRATI stack and that the actual IPC module is loaded. The following Listings show the scripts.

Listing C.6: VM1: setup-vm1.sh

```
#!/bin/bash
ip link add link eth1 name eth1.100 type vlan id 100
ip link set dev eth1 up
ip link set dev eth1.100 up

ip link add link eth2 name eth2.400 type vlan id 400
ip link set dev eth2 up
ip link set dev eth2.400 up

modprobe shim-eth-vlan
modprobe rina-default-plugin
modprobe normal-ipcp
```

Listing C.7: VM2: setup-vm2.sh

```
#!/bin/bash
ip link add link eth1 name eth1.100 type vlan id 100
ip link set dev eth1 up
ip link set dev eth1.100 up

ip link add link eth2 name eth2.200 type vlan id 200
ip link set dev eth2 up
ip link set dev eth2.200 up

modprobe shim-eth-vlan
modprobe rina-default-plugin
modprobe normal-ipcp
```

Listing C.8: VM3: setup-vm3.sh

```
#!/bin/bash
ip link add link eth1 name eth1.200 type vlan id 200
ip link set dev eth1 up
ip link set dev eth1.200 up

ip link add link eth2 name eth2.500 type vlan id 500
ip link set dev eth2 up
ip link set dev eth2.500 up

ip link add link eth3 name eth3.300 type vlan id 300
ip link set dev eth3 up
ip link set dev eth3.300 up

modprobe shim-eth-vlan
modprobe rina-default-plugin
modprobe normal-ipcp
```

Listing C.9: VM4: setup-vm4.sh

```
#!/bin/bash
ip link add link eth1 name eth1.400 type vlan id 400
ip link set dev eth1 up
ip link set dev eth1.400 up

ip link add link eth2 name eth2.300 type vlan id 300
ip link set dev eth2 up
ip link set dev eth2.300 up

modprobe shim-eth-vlan
modprobe rina-default-plugin
modprobe normal-ipcp
```

Listing C.10: VM5: setup-vm5.sh

```
#!/bin/bash
ip link add link eth1 name eth1.500 type vlan id 500
ip link set dev eth1 up
ip link set dev eth1.500 up

modprobe shim-eth-vlan
modprobe rina-default-plugin
modprobe normal-ipcp
```

Appendix D

Enrolling DIFs

In both experiments, DIFs are enrolled in a special order to connect nodes to each other. In this appendix the enrollment codes are described and the IPC process lists are shown. When the IRATI Demonstrator is used, the enrolling of DIFs is done automatically by the provided scripts. To run the *enroll-to-dif* command, one needs to connect to the IPC Console using the following command: *socat - UNIX:/var/run/ipcm-console.sock*.

Listing D.1: Enrolling DIFs

```
# vm3
enroll-to-dif 4 providerBRegional.DIF 200 providerBRegional.2.IPCP 1
enroll-to-dif 5 providerARegional.DIF 300 providerARegional.4.IPCP 1

# vm1
enroll-to-dif 3 providerBAccess.DIF 100 providerBAccess.2.IPCP 1
enroll-to-dif 4 providerAAccess.DIF 400 providerAAccess.4.IPCP 1

# vm3
enroll-to-dif 6 upstream.DIF 500 upstream.5.IPCP 1
enroll-to-dif 7 providerAInternet.DIF providerARegional.DIF providerAInternet.4.IPCP 1

# vm1
enroll-to-dif 5 providerAInternet.DIF providerAAccess.DIF providerAInternet.4.IPCP 1

# vm5
enroll-to-dif 3 providerAInternet.DIF upstream.DIF providerAInternet.3.IPCP 1

# vm3
enroll-to-dif 8 providerBInternet.DIF upstream.DIF providerBInternet.5.IPCP 1

# vm2
enroll-to-dif 5 providerBInternet.DIF providerBRegional.DIF providerBInternet.3.IPCP 1

# vm1
enroll-to-dif 6 providerBInternet.DIF providerBAccess.DIF providerBInternet.2.IPCP 1
```

D.1 IPC Process Lists

This section shows the IPC process lists per VM. These lists are the same for both experiments. As shown below, the different DIFs that are stacked on top of each other are registered to the DIF below. The eth DIFs are the special shim DIFs that are connected over a VLAN interface for

connecting the VMs to each other. To run the *list-ipcps* command, one needs to connect to the IPC Console using the following command: *socat - UNIX:/var/run/ipcm-console.sock*.

Listing D.2: IPC Process List for VM1

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications | Port-ids of
    flows provided)
    1 | eth.1.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 100 | providerBAccess.1.IPCP-1-- |
        1
    2 | eth.2.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 400 | providerAAccess.1.IPCP-1-- |
        2
    3 | providerBAccess.1.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerBAccess.DIF |
        providerBInternet.1.IPCP-1-- | 4
    4 | providerAAccess.1.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerAAccess.DIF |
        providerAInternet.1.IPCP-1-- | 3
    5 | providerAInternet.1.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerAInternet.DIF |
        - | -
    6 | providerBInternet.1.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerBInternet.DIF |
        - | -
```

Listing D.3: IPC Process List for VM2

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications | Port-ids of
    flows provided)
    1 | eth.1.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 100 | providerBAccess.2.IPCP-1-- |
        2
    2 | eth.2.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 200 | providerBRegional.2.IPCP-1--
        | 1
    3 | providerBRegional.2.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerBRegional.DIF |
        providerBInternet.2.IPCP-1-- | 3
    4 | providerBAccess.2.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerBAccess.DIF |
        providerBInternet.2.IPCP-1-- | 4
    5 | providerBInternet.2.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerBInternet.DIF |
        - | -
```

Listing D.4: IPC Process List for VM3

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications | Port-ids of
    flows provided)
    1 | eth.1.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 200 | providerBRegional.3.IPCP-1--
        | 1
    2 | eth.2.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 300 | providerARegional.3.IPCP-1--
        | 2
    3 | eth.3.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 500 | upstream.3.IPCP-1-- | 3
    4 | providerBRegional.3.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerBRegional.DIF |
        providerBInternet.3.IPCP-1-- | 7
    5 | providerARegional.3.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerARegional.DIF |
        providerAInternet.3.IPCP-1-- | 4
    6 | upstream.3.IPCP:1:: | normal-ipc | ASSIGNED TO DIF upstream.DIF | providerAInternet
        .3.IPCP-1--, providerBInternet.3.IPCP-1-- | 5, 6
    7 | providerAInternet.3.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerAInternet.DIF |
        - | -
    8 | providerBInternet.3.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerBInternet.DIF |
        - | -
```

Listing D.5: IPC Process List for VM4

```
IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications | Port-ids of
    flows provided)
    1 | eth.1.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 300 | providerARegional.4.IPCP-1--
        | 1
    2 | eth.2.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 400 | providerAAccess.4.IPCP-1-- |
        2
    3 | providerARegional.4.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerARegional.DIF |
        providerAInternet.4.IPCP-1-- | 3
    4 | providerAAccess.4.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerAAccess.DIF |
        providerAInternet.4.IPCP-1-- | 4
    5 | providerAInternet.4.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerAInternet.DIF |
        - | -
```

Listing D.6: IPC Process List for VM5

```
IPCM >>> enroll-to-dif 3 providerAInternet.DIF upstream.DIF providerAInternet.3.IPCP 1
DIF enrollment succesfully completed

IPCM >>> list-ipcps
Current IPC processes (id | name | type | state | Registered applications | Port-ids of
    flows provided)
    1 | eth.1.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 500 | upstream.5.IPCP-1-- | 1
    2 | upstream.5.IPCP:1:: | normal-ipc | ASSIGNED TO DIF upstream.DIF | providerAInternet
        .5.IPCP-1--, providerBInternet.5.IPCP-1-- | 2, 3
    3 | providerAInternet.5.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerAInternet.DIF |
        - | -
    4 | providerBInternet.5.IPCP:1:: | normal-ipc | ASSIGNED TO DIF providerBInternet.DIF |
        - | -
```

Appendix E

Memory Script

This appendix describes the script that was used to save the available RAM of the VM to a log file. This script was automatically launched during boot of the VMs in the IRATI Demonstrator and the VMs with the compiled IRATI stack.

Listing E.1: memory.sh

```
#!/bin/sh
i=0
while [ 1 ]
do
        echo $i" "‘free -m | awk ’/^Mem/ {print $4}’‘ >> /var/log/memory.log
        i=$((i+1))
        sleep 1
done
```