Recursive InterNetworking Architecture (RINA)
Boston University Prototype
Programming Manual
(version 1.0)

Yuefeng Wang Flavio Esposito Ibrahim Matta John Day
{wyf, flavio, matta, day}@bu.edu

Recursive InterNetworking Architecture Laboratory
Computer Science Department
Boston University

`http://csr.bu.edu/rina`

Technical Report BUCS-TR-2013-013

Last update: November 8, 2013

Contents

Acknowledgement

We would like to express our gratitude to other (current and past) members of the RINA team, not only from Boston University but also our collaborators from TRIA Network Systems and IRATI Project. Thanks for their support and encouragement, especially Professor Lou Chitkushev. Also we would like to thank Yue Zhu for his implementation of the video streaming application, which is a great example demonstrating the usage and advantages of our RINA prototype.

Lastly we would like to thank the National Science Foundation (NSF grant # CNS-0963974) and Global Environment for Network Innovations (GENI) Project Office. This work would not have been possible without their support.

Chapter 1

Introduction

The lack of a complete Internet architecture continues to dramatically increase the management costs and complexity of all distributed systems. For example, (1) supporting mobile devices (nodes), that change their network connection as they move, has been problematic because of a deficient addressing structure that identifies a node with a particular network connection; (2) distinguishing applications with public static identifiers make them exposed and vulnerable to attacks; and (3) treating the whole Internet as a single homogeneous network that exposes a rudimentary "best-effort" delivery service makes it impossible to fully exploit advanced capabilities of new network technologies.

Fundamentally, the problem with the current Internet architecture is that a layer expects a certain level of service from the underlying layer, but the underlying layer may not meet that requirement. This inability of layers to communicate what service is needed, together with the desire to offer virtualized network services, have compounded existing network service management challenges. These problems are unique to the Internet and are not found in other network architectures of the past, such as CYCLADES [1], XNS [2], DECNET [3], or OSI [4]. In response to such challenges, the research community has directed significant efforts and resources toward clean slate architecture design [5, 6, 7, 8]. We contribute to the efforts of redesigning the Internet with a Recursive InterNetwork Architecture (RINA) [9].

RINA is based on the fundamental principle that networking is Inter-Process Communication (IPC). RINA recurses the IPC service over different scopes. Specifically, a scope defines a Distributed IPC Facility (DIF) comprised of the set of IPC processes, possibly running on different machines, that collaboratively provide a set of well-defined flow services to upper application processes. The mechanisms within each IPC process are the same but each DIF layer is instantiated using a possibly different set of policies.

Following the handbook of RINA specification [10], we implemented a prototype, whose code is available at [12]. The prototype is tested on our BU campus network, and cross-debugged with two other RINA prototypes (IRATI [13] and TRIA [14]). Also we have demonstrated our prototype on the GENI testbed [15]. This document presents a programmer manual and a user manual for the prototype. We have been adding new components to our prototype for over two years. The current version [12] resulted in about $50,000$ lines of Java code, excluding support libraries, test applications, and configuration files. This manual provides details of the software architecture of our prototype, and guides users on how to use existing policies, and write new networking applications leveraging the provided mechanisms. It should be noted that RINA (both here and elsewhere [13, 14]) is still "in progress." This implementation does not yet fully interwork with the other RINA implementations and all are being extended to implement the full complement of the specifications. Modifications should be expected.

After reading the manual, programmers should know how to create new or modify existing configuration files, and how to add new policies using the existing interfaces. We hope that with this effort, the networking and distributed system communities will actively benefit from the principles proposed by the RINA architecture, developing new applications by leveraging our released prototype.

In Chapter 2 we review some of the key and most recent representative tools, and explain the contributions of our released RINA prototype in context. In Chapter 3 we give a brief design overview of the main RINA architecture components and mechanisms, while in Chapter 4 we describe the details of our prototype implementation. In Chapter 5, we walk-through a researcher on how to edit the RINA configuration file. We also describe how a research would write their own application leveraging our API and our publish/subscribe mechanism. As an example of how to set up a RINA application, we then describe how to reproduce a video streaming demonstration using the service provided by a DIF. We then conclude this manual with instructions on how to add additional management policies to RINA. In particular, we focus on authentication, routing and addressing policies in Chapter 6.

Chapter 2

Related Work

The networking and distributed system community is becoming skeptical of the significance of research results based solely on simulations. This fostered extensive research oriented towards networked simulation, emulation, and prototype tools, in order to help perform realistic but reproducible (wide-area) network experiments. These tools can be classified into simulators, emulators, simulators with emulated interfaces, and experimental protocols with interfaces for rapid prototyping.

2.1 Existing Network Experimental Tools

Mininet is an open-source network emulator [21], developed to foster rapid prototyping in the context of Software-Defined Networking (SDN). With the help of the Mininet API, researchers can test their own protocols or architectures on emulated virtual graphs. It is possible in fact to create realistic virtual networks with hosts, switches, routers and links, reserving virtual node and link capacity. All virtual networks are contained into a single hosting machine, and each Mininet virtual node can run real (Linux) applications such as `ping`, `traceroute` or `iperf` [22]. Mininet enables experimenters to customize their network graph, including CPU and bandwidth constraints on the virtual nodes and links, and allows multiple developers to work on the same graph at the same time. Mininet also supports OpenFlow [28] switches for customized routing and SDN applications. One of the most valuable features of Mininet is however its support for standard Linux network API, and its Command Programming Interface (CLI), a Python API for rapid network creation and experimentation.

As Mininet, our prototype also provides an interface for rapid networked application prototyping, but without limiting the scopes of the experiments to a single physical machine. RINA experiments can run on real networks, for example within the GENI testbed [33], an it is currently running on other testbeds in Europe [13] and over the internet, as long as there is connectivity among the prototype nodes.

Click is an open-source (both user- and kernel-space) tool that can be leveraged to build software for programmable and configurable routers [23]. Click configurations are modular and easy to extend, thus providing a flexible platform for network protocol development, testing and deployment. A Click router is composed of packet processing modules called *elements*. Such elements enable a fine-grained control over the forwarding path. A Click router configuration is a directed graph that use such elements as vertices. Each element implements a simple router function. A possible path of packets is represented by *connections* between pairs of elements, and packets flowing along the edges of such graph. The Click implementation supports different packet scheduling and dropping policies, several queuing policies, and other packet processing applications [24].

Our RINA prototype has been released as a rapid prototyping tool as Click, but our focus is not limited to a single router. Applications built using our prototype may use the Linux system architecture but the current version does not support kernel functionalities. The RINA community is currently building a kernel prototype of the architecture [13].

Network Simulator (ns-3) advertises itself as an open source discrete event network simulator, which allows studying Internet protocols and large-scale systems in a controlled environment for research and educational purposes [17]. Unfortunately ns-3 is not backwards-compatible with its widely diffused predecessor ns-2 [16]. Ns-3 design focus has been on improving the core architecture of ns-2, enabling integration with other software for both research and educational applications. Ns-3 maintains an extensible software core, and has a flexible tracing system. Its implementation supports interfaces such as sockets API, and Linux device driver interface. It also supports better software integration than ns-2, allowing integration with other open source tools and models. For example, ns-3 has an interface to the Click modular router [24], and an interface to OpenFlow [28, 29]

to allow the simulation of virtual routers and switches. Ns-3 supports two modes of integration with real systems: (1) virtual machines running on top of ns-3 communication channels, and (2) an ns-3 stack over real devices.

We share ns-3 design goal of releasing a tool valid for both research and teaching purposes, but ours is not a simulation tool. Moreover, our design goal has been to ferret out fundamental principles and understand their implications for network architecture, not an improvement of a previous version of the tool. Also, our RINA prototype currently does not support any integration with existing tools like Click or Openflow.

Optimized Network Engineering Tools (OPNET) is a commercial network simulator that provides a virtual network environment able to model an complete network, including hosts, routers, switches, protocols, and applications [25]. OPNET is a commercial software, but is free to qualifying universities worldwide for academic research and teaching [26]. OPNET provides a hierarchical GUI-based editor (network, node and process), and supports discrete event simulation and flow analysis: a discrete event simulation allows packets and protocol message simulations, and the flow analysis enables modeling steady-state network behavior. OPNET also provides interfaces for live integration with network system and other simulators [27].

As in OPNET, with our RINA prototype researchers are also able to model a complete network, including hosts and routers. Our prototype however, does not require (and does not suggest) that users write additional protocols, but merely to instantiate the policies of existing control and data management mechanisms. Users may write their own applications (objects) and use our API to operate on their states.

OpenFlow is an open standard which enables researchers to run experimental protocols [28]. It relies on Ethernet switches, and provides standardized interfaces to add and remove flow entries on switch's internal flow-table [29]. With OpenFlow, researchers can run experiments without knowing the internal details of vendor's network devices. An OpenFlow switch separates packet forwarding (data plane) and routing decisions (control plane). The data plane still resides on the switch, but the control plane is moved to a separate (OpenFlow) controller. The OpenFlow switch and controller communicate in user space through the OpenFlow protocol [28].

As in Openflow, with our RINA prototype researchers can also instantiate policies for the forwarding mechanism, in support of SDN research, without requiring a centralized controller to inject the forwarding rules [1]. With our tool however, researchers are able to instantiate other (data transfer and management) mechanisms as well, such as routing or authentication. In other words, with RINA we can define policies for software-defined networks, not merely software-defined switches.

2.2 Unique Features of our RINA Prototype

Our RINA prototype can be used as a network tool enabling researchers to develop and test their own protocols and networking applications. Researchers can leverage our API and use the prototype to test the existing supported policies in their own customized environment, they can add novel policies, or leverage the RINA communication service to write external applications. Our prototype can also be used as a teaching tool for system undergraduate and graduate classes, to help students understand basic concepts, or perform advanced research in the field of computer networking and distributed systems.

Customizable Scope: a DIF can be quickly and easily set up and customized by leveraging the provided mechanisms of our framework. Configurable policies include instantiation of several management mechanisms, $e.g.$ routing or enrollment. None of the existing simulation and emulation tools enable researchers to test different routing policies in different private networks (DIFs), each within a given limited scope.

Policy-based Networking: our RINA architecture supports Software Defined Networking (SDN). Users can set up a private virtual network, merely as a high level DIF or Distributed Application Facility (DAF) (Section 3.1) on top of existing physical networks (lower level DIFs), and instantiate a variety of mechanisms, rather than just forwarding as in OpenFlow.

Recursion: another feature that makes the RINA architecture unique in the space of testing tools is recursion. The same mechanisms ($e.g.$ transport, routing, and other management mechanisms) are repeated over different scopes and potentially instantiated with different policies. Both data and control traffic is aggregated and transferred to lower IPC layers, and each (private) DIF regulates and allocates resources to user traffic originating from higher-level DIFs or applications.

[1] Recent research has shown how we might need more than one controller [32].

Chapter 3

Understanding RINA

The very basic premise of this architecture, yet fresh perspective, is that networking is not a layered set of different functions but rather a single layer of distributed Inter-Process Communication (IPC) that repeats over different scopes. Each instance of this repeating IPC layer implements the same functions / mechanisms but policies are tuned to operate over different ranges of the performance space (e.g., capacity, delay, loss).

As shown in Figure 3.1, Distributed Application Processes (DAPes) communicate via an Inter-Process Communication (IPC) facility and they can themselves be IPCs (hence the recursion). Since the facility is distributed, we call an instance of a RINA network a Distributed IPC Facility (DIF), and DIF is a collection of IPC processes providing communication service. This section is intended to give a brief overview of our RINA architecture.

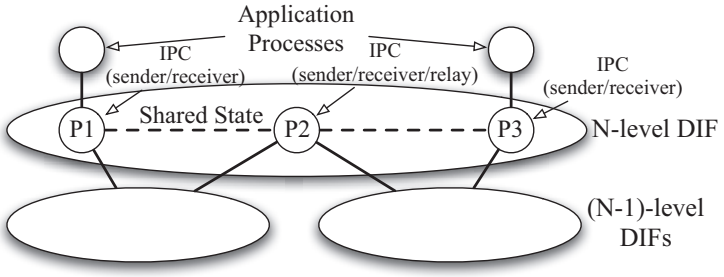

Figure 3.1: Example of a RINA Network: RINA has one building block (the DIF layer) that can be composed to form a higher-level DIF from the services provided by lower-level DIFs. This figure illustrates composition by recursion of the DIF layer.

3.1 Distributed IPC Facility (DIF) and Distributed Application Facility (DAF)

The DIF is a service building block that can be repeated and composed in layers to build wider scoped services that meet user requirements. A DIF can be thought of as a private network and it is different from the traditional definition of layer in the Transmission Control Protocol/Internet Protocol (TCP/IP) architecture. First, a DIF does not perform a single function or small subset of pre-determined functions, but a coordinated set of policy managed functions to achieve the desired IPC service. Second, the DIF naturally separates various concerns, including operation over different time scales ($e.g.$ short-term data transfer and multiplexing versus long-term connection management and access control issues).

More generally, we call a set of Distributed Application Processes (DAPes) cooperating to perform a certain function a Distributed Application Facility (DAF). This function can be communication service, management service or any other service. A DIF is a specific DAF whose job is only to provide communication service.

3.2 IPC Architecture and Mechanisms

With the concept of Inter-Process Communication (IPC), the architecture refers to the general model of communicating processes. However, in this document we use the same term, interchangeably with the term IPC Process, to refer to a specific implementation of it as well. Each application process that needs to establish a communication flow, has to do it through an underlying IPC process via a DIF. Each IPC process consists of three distinct sets of tasks dealing with IPC aspects at different computation timescales (Figure 3.2). These task sets are loosely coupled through a Resource Information Base (RIB), which is a database in every process, and it stores all the information useful for the management of a DIF or DAF.

These three tasks are as follows:

1. an *IPC Data Transfer*, which supports relaying (forwarding), multiplexing (scheduling), and per-flow data transfer (computation is simple and therefore fast),
2. an *IPC Transfer Control*, which implements the error and flow control, and controls the per-flow data transfer parameters,
3. an *IPC Management*, which implements the Common Distributed Application Protocol (CDAP) to query and update a RIB for routing, security, resource management, address assignment, and so on (computation is complex and therefore slow).

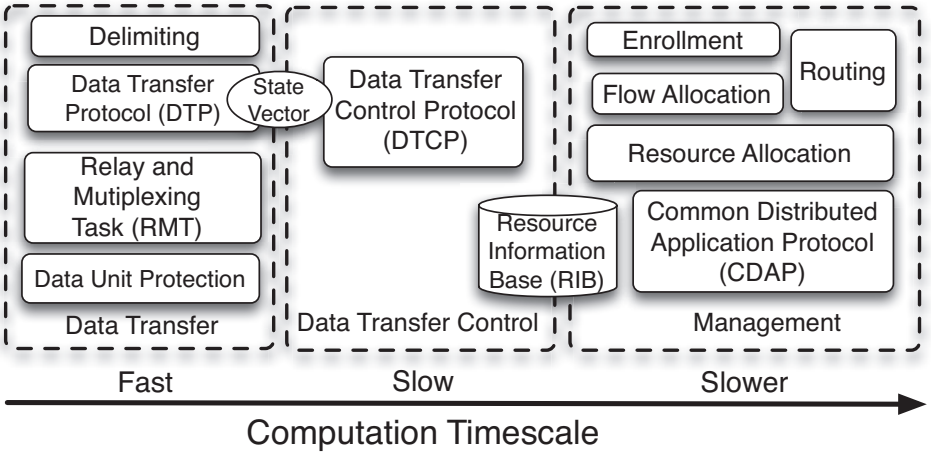

Figure 3.2: Architecture of an IPC: each IPC process consists of three distinct sets of mechanisms dealing with IPC aspects at different timescales: IPC data transfer, IPC data control and IPC management.

3.3 Inter-DIF Directory (IDD) Service

The Inter-DIF Directory (IDD) service is an important component of the RINA architecture. It is responsible for naming management by answering queries via a set of local agents responsible for intra-DIF name resolution. The IDD service is used by an IPC process or by an application process to locate a service. Such service can be either a DIF providing communication service, or a DAF providing a certain functionality.

3.4 The Protocols

3.4.1 Common Distributed Application Protocol (CDAP)

The Common Distributed Application Protocol (CDAP), inspired and modeled after CMIP, an object-based standards-defined protocol [31], is used by communicating RINA applications to exchange any structured application-specific data necessary to coordinate DIFs. CDAP is envisioned to be used in either a request/response mode or a subscribe/publish mode. A

CDAP entity (Management Application Entity) collects information from other peers in the DIF in response to events ($e.g.$, failures) in the network, periodically, or as a matter of policy imposed by its users. Some of these events may involve all members of the DIF or be more limited in scope.

3.4.2 Error and Flow Control Protocol (EFCP)

The Error and Flow Control Protocol (EFCP) is the data transfer protocol required to maintain an instance of IPC within a DIF. The functions of this protocol ensure reliability and flow control as required. The original design of the Error and Flow Control Protocol (EFCP) is modeled after Richard Watson's Delta-t transmission protocol [34], and includes a Data Transport Protocol (DTP) and a Data Transport Control Protocol (DTCP).

3.5 Enrollment Procedure

As we mentioned earlier, a DIF consists of a collection of IPC processes. An IPC process has to be enrolled into a DIF by one existing member in order to communicate with other IPC processes inside the DIF. An enrollment procedure creates, maintains, distributes (and eventually deletes upon withdrawal) information within a DIF. Such information may include addressing, access-control rules or other management policies necessary to create instances and characterize a communication. In this section, we walk through the details of the enrollment procedure.

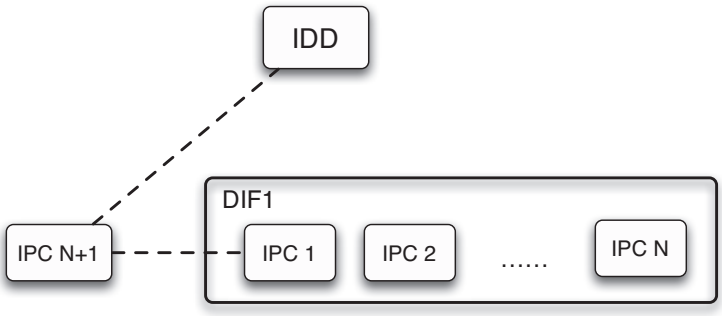

Figure 3.3: IPC N+1 wants to join DIF1 which contains N IPC processes.

We model a case in which a DIF, *DIF 1* is formed by N IPC processes, and another *IPC N+1* enrolls to join *DIF 1*. The enrollment procedure starts with an *IPC N+1* request for *DIF 1* to the IDD process. When the IDD process receives such request, a database lookup returns a list of IPC processes, together with their supporting DIF information (Figure 3.3).

When *IPC N+1* obtains the reply, it learns that *IPC 1* is the only reachable member of *DIF 1*, based on the supporting DIF information. This means that *IPC N+1* and *IPC 1* share a supporting (underlying) DIF. Only after this discovery operation, *IPC N+1* can send an *M_CONNECT* message containing some authentication information to *IPC 1*. If the received authentication information is valid, *IPC1* returns an *M_CONNECT_R* with a positive result to *IPC N+1*. [1] This message informs *IPC N+1* that *IPC1* is ready to start the enrollment. *IPC N+1* sends an M_START to *IPC N+1*, and after receiving it, *IPC 1* replies with an M_START_R, which contains a dynamically generated address (private to *DIF 1*) for the new member *IPC N+1*.

After obtaining such address, the enroller *IPC 1* process sends a sequence of *M_CREATE* messages to *IPC N+1* which contain information about the current members in the DIF and the applications reachable through this DIF. Upon completion, *IPC 1* sends an *M_STOP* to *IPC N+1* indicating the termination of all the information required for the enrollment, and *IPC N+1* replies with an *M_STOP_R* to *IPC1*. After receiving an *M_STOP_R*, *IPC 1* sends an *M_START* to *IPC N+1* to confirm that the enrollment procedure is completed. *IPC N+1* is now able to operate in *DIF 1*. Finally, *IPC 1* sends an *M_CREATE* message to other existing IPC processes (*IPC 2 . . . IPC N*), to inform them about the new enrolled member.

This Enrollment procedure is designed to be used in an environment where IPC Processes might disappear for short periods of time and reappear, i.e. router crashes. It is expected that other enrollment procedures will be defined for other environments.

[1] Each CDAP message contains a "result" field, that, if positive, communicates a successful operation on any of the management objects, else an error code.

Chapter 4

Implementing RINA

4.1 Software Architecture

In this section we present an overview of the main packages comprising the RINA prototype. Logically, we can divide the software architecture into the following components:

- An Inter-Process Communication (IPC) set of packages, which logically includes:
  - An IPC interface and implementation package
  - A Resource Information Base (RIB) interface and implementation package
  - A RIB Daemon interface and implementation package
  - An IPC Resource Manager (IRM) interface and implementation package
  - An Application Entity (AE) package, including the Management AE and Data Transfer AE
  - A Flow Allocator (FA) interface and implementation package
  - A Routing Daemon implementation package
- A message package that supports the implementation of Error and Flow Control Protocol (EFCP) message format, $i.e.$ Data Transfer Protocol (DTP) and Data Transfer Control Protocol (DTCP), and of Common Distributed Application Protocol (CDAP)
- An Inter-DIF Directory (IDD) package, external to the architecture of an IPC process but used by IPC processes to locate application services
- A RINA object package containing all objects used in the current architecture
- A shim layer package where TCP connections are used to emulate physical connectivity
- A Distributed Application Process (DAP) package useful for programmers to understand how to implement their own applications using the existing code
- A RINA Node package implementing the RINA node process where application processes and IPC processes reside
- An input /output configuration package

We show our software architecture on a *RINA node* in Figure 4.1, where an application process uses the communication service provided by an N-level IPC process, and recursively an N-level IPC process uses the communication service provided by an (N-1)-level IPC process. A TCP shim layer at the bottom emulates physical connectivity.

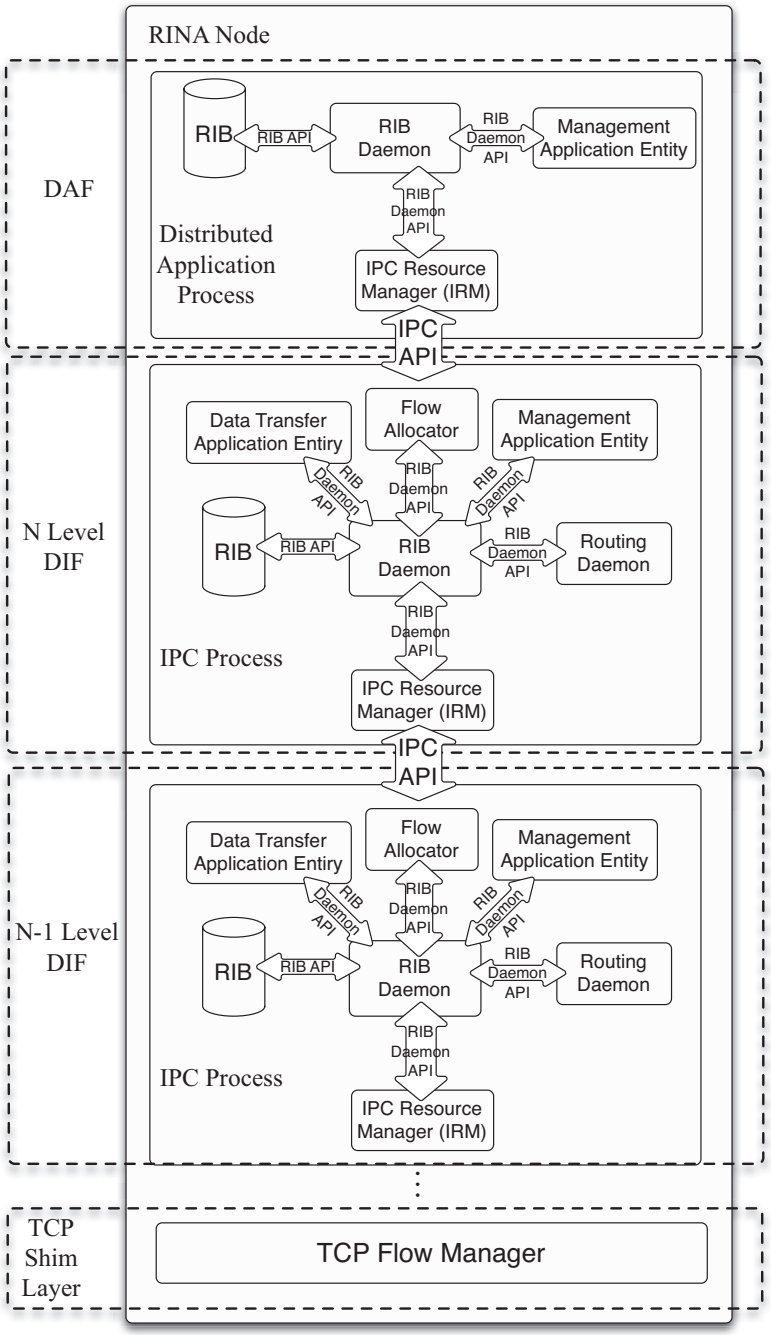

Figure 4.1: RINA Node overview.

With the goal of helping programmers navigate through the implementation, speeding up their understanding of our prototype, in the rest of this chapter we explain the details of our implementation.

4.2 IPC Related Packages

In this section, we explain details of all packages related to the IPC process.

4.2.1 IPC Package

(1) IPC Interface

An IPC API is used by an application process to access an IPC process: (1) via the IPC's Flow Allocator to create and delete a flow to another application process, via Data Transfer AE to send and receive messages over an existing flow, (2) to register itself to the underlying DIFs such that it can be reached through the IPC process. Recursively, an N-level IPC process is seen as an application process which uses an (N-1)-level IPC process's communication services. Details of the *Flow* object are explained in Section 4.5.

IPC Process Interface

public int allocateFlow(Flow flow);
public void deallocateFlow(int portID);
public void send(int portID, byte[] msg) throws Exception;
public byte[] receive(int portID);

public void registerApplication(ApplicationProcessNamingInfo apInfo, FlowInfoQueue flowInfoQueue); public void deregisterApplication(ApplicationProcessNamingInfo apInfo);

Table 4.1: Java IPC interface provided to application process.

(2) IPC Implementation

When an IPC process is instantiated, it first reads from its configuration file to bootstrap its components based on different policies. Every IPC process consists of the following components:

(1) **Resource Information Base (RIB)**: this component is responsible for storing all information $e.g.$, routing updates, useful for the management of a DIF (Section 4.2.2).

(2) **RIB Daemon**: this component is responsible for managing the information stored in the RIB and its veracity, updating and making states available to (IPC and application) processes (Section 4.2.3).

(3) **IPC Resource Manager (IRM)**: this component is responsible for managing the use of (N-1)-DIFs to allocate and maintain the connections between the N-level IPC process and its peer IPC processes within the DIF. Recursively, an N-level IPC's IRM uses the (N-1)-level IPC's API, where a connection at N-level is mapped to an (N-1)-level flow maintained by the (N-1)-level IPC process' Flow Allocator (Section 4.2.4).

(4) **Management Application Entity (Management AE)**: this component is responsible for the enrollment of the IPC process when joining a DIF. It also handles all CDAP messages received from other IPC processes' management AEs (Section 4.2.5).

(5) **Data Transfer Application Entity (Data Transfer AE)**: this component is responsible for data transfer (implementing Error and Flow Control Protocol ($i.e.$ EFCP) for each existing flow, and also responsible for the Relay and Multiplexing Task (RMT) (Section 4.2.5).

(6) **Flow Allocator (FA)**: this component is responsible for the flow allocation API invocation from application processes, and maintains every flow allocated (Section 4.2.6).

(7) **Routing Daemon**: this component is responsible for the routing inside the DIF, and it is initiated by the Management AE after the IPC process is enrolled into a DIF (Section 4.2.7).

4.2.2 RIB Package

Every (IPC and application) process stores all its information inside the Resource Information Base (RIB). Attributes of the RIB can be created, read or written through the RIB interface. When some information is needed, it is always retrieved through the RIB.

(1) RIB Interface

The RIB interface allows adding/reading/removing an attribute to/from the RIB, and also adding/removing an IPC member into/from the DIF member list when a new member joins the DIF or an existing member leaves.

RIB Interface
public void addAttribute(String attributeName, Object attribute);
public Object getAttribute(String attributeName);
public void removeAttribute(String attributeName);
public int addMember(String ipcName, String ipcInstance, LinkedList<String> underlyingDIFs);
public void removeMember(int address);

Table 4.2: Java RIB Interface.

(2) RIB Implementation

The attributes of the RIB are stored in a hashed data structure where the key is the attribute name and the value is an Object class object, to be cast into any Java object when retrieved from the RIB. For convenience, RIB stores all information related to the members of the DIF separately.

4.2.3 RIB Daemon Package

Every (IPC and application) process has a RIB daemon. Different states of the DIF may need different information that may be updated with different frequency or upon the occurrence of some event. Based on a Pub/Sub model, the RIB daemon provides such service. A subscription event is mapped by the RIB daemon which recognizes the objects from their type, and acts upon different requests with a set of operations on objects in a local or remote RIB.

A local (read/write) operation may consist of a direct access to attributes of objects in a local RIB, while an infrequent and wide-area operation on remote objects is supported by generating a sequence of CDAP messages to remote (IPC or application) processes' RIBs.

(1) RIB Daemon Interface

The RIB Daemon interface allows the creation and deletion of a **Pub/Sub** event, and retrieving of information through a **Sub** event, or publishing information through a **Pub** event. When a new event is created, an event ID is returned to the subscriber. Details of the *SubscriptionEvent* object are explained in Section 4.5.

RIB Daemon Interface
public int createEvent(SubscriptionEvent subscriptionEvent);
public void deleteEvent(int subscriptionID);
public Object readSub(int subID);
public void writePub(int pubID, byte[] obj);

Table 4.3: Java RIB Daemon Interface.

(2) RIB Daemon Implementation

The RIB daemon maintains all subscription events, including **Sub** events and **Pub** events. For each subscription event, the RIB daemon creates a corresponding event handler (defined in `rina.ribDaemon.util/EventHandler.java`) which is responsible for updating the event. For each **Sub** event to remote processes' RIBs, the event handler translates the event to a CDAP message (M_CREATE), which contains a subscription event object and sends it to the remote process's management AE. For each **Pub** event, the event handler creates a publisher (defined in `rina.ribDaemon.util/Publisher.java`) to periodically publish information to its subscribers.

When a Management AE receives a **Sub** object from another process, it adds the sender to the subscriber list of a predefined **Pub** event (if any). When a Management AE receives a **Pub** object from another process, it updates the corresponding **Sub** event.

4.2.4 IPC Resource Manager (IRM) Package

On behalf on the application process or N-level IPC process, the IPC Resource Manager (IRM) component manages the use of the (N-1)-DIFs to allocate and maintain the connections between the N-level IPC process and its peer IPC processes inside the DIF, or the connections between the Application Process and its peer Application Processes inside the Distributed Application Facility (DAF).

(1) IRM Interface

The IRM interface allows for allocating/deallocating a connection, or sending/receiving data over an existing connection. When a new connection is created, a handle ID is returned to the caller, then the handle ID is used to send and receive data.

IRM Interface
public int allocateFlow(Flow flow)
public void deallocateFlow(int handleID);
public void send(int handleID, byte[] msg) throws Exception;
public byte[] receive(int handleID);

Table 4.4: Java IRM Interface.

(2) IRM Implementation

Each DIF-0 IPC process's IRM has a component called **WireManager** (defined in `rina.irm.util/WireManager`) which manages the use of the physical wires, and DIF-0 connections are mapped on physical wires. When the **allocateFlow** method is called, the IRM uses WireManager to set up the DIF-0 connections over the wire. The handle returned by the **allocateFlow** method corresponds to a wire and port number on this wire. In our current implementation, the wires are emulated by TCP connections, so for DIF-0 IPC processes, their underlying DIF is the TCP shim layer (Section 4.6). The WireManager directs all method calls from IRM to the TCP Flow Manager (defined in `rina.tcp/TCPFlowManager`).

For non-zero DIF IPC processes and application processes, their underlying supporting DIF is a regular DIF. When the **allocateFlow** method is called, the IRM first finds an underlying IPC process, through which the destination process can be reached, then it uses the IPC interface of the underlying IPC process to allocate a flow. The handle returned by the **allocateFlow** method corresponds to an underlying IPC process and a port number on that IPC process. If no such IP underlying IPC process can be found, a new DIF can be dynamically formed to serve this allocation request.[1]

IRM maintains all existing connections. It creates a *HandleEntry* object (defined in `rina.irm.util/HandleEntry.java`) for each connection, and the *HandleEntry* object contains all information about the connection, including handle ID, source application information, destination application information, underlying IPC process (or wire ID) and port number.

4.2.5 RINA Application Entity Package

Each IPC process has two Application Entities (AE), Management AE (defined in `rina.ipc.ae/ManagementAE.java`) and Data Transfer AE (`rina.ipc.ae/DataTransferAE.java`). An application entity is identified by Application Process Name, Application Process Instance, Application Entity Name and Application Entity Instance.

(1) Management Application Entity (AE)

Depending on its configuration file, when an IPC process is initiated, it can be either the first member of a DIF or a member waiting to join a DIF. The Management AE is responsible for the enrollment of the IPC process when joining a DIF.

[1] See Appendix A for details about the flow allocation.

The Management AE is able to enroll new IPC processes into the DIF by attaching a Management AE Handler (defined in `rina.ipc.ae/ManagementAEHandler.java`) to each new IPC member.

The Management AE Handler handles all CDAP messages received from other IPC processes' Management AEs. Also when an application is using this IPC process as an underlying IPC, the Management AE registers this application by sending registration information to every member inside the DIF, so that every IPC process in the DIF becomes aware that the application is reachable through itself. The Routing Daemon (Section 4.2.7), which is responsible for the routing inside the DIF, is started once the IPC process joins a DIF.

(2) Data Transfer Application Entity (AE)

The Data transfer AE is responsible for data transfer for each existing flow, and the Relay and Multiplexing Task (RMT). Error and Flow Control Protocol (EFCP) is the protocol for data transfer in RINA architecture, and it includes Data Transfer Protocol (DTP) and Data Transfer Control Protocol (DTCP). In our current implementation, only a simple version of DTP is supported, and DTCP is not yet supported.

The Data transfer AE creates a Data Transfer AE handler (defined in `rina.ipc.ae/DataTransferAEHandler.java`) for each connection between itself and the other IPC process's Data transfer AE, and it relays and multiplexes Data Transfer Protocol (DTP) packets that encapsulate DTP packets or CDAP packets from application processes.

When a DTP packet is received, the Data Transfer AE handler inspects its header information, and it delivers it to the corresponding application process using this IPC process based on the connection ID if the IPC process is the destination. If the destination is not itself, it checks the forwarding table and sends the packet to the next-hop toward the destination.

4.2.6 Flow Allocator Package

(1) Flow Allocator Interface

The Flow Allocator (FA) interface is used when an IPC process gets a flow allocate request from an application process through the IPC Interface or when the IPC process receives a flow allocate request from another IPC process's flow allocator. So it can (1) create a flow for an application process using this IPC as underlying IPC, (2) create a flow upon the request from another IPC process's flow allocator, (3) delete an existing flow. Details of the *Flow* object are explained in Section 4.5.

Flow Allocator Interface
public int submitAllocationRequest(Flow flow);
public int receiveAllocationRequest(Flow flow);
public void deallocateFlow(int portID);

Table 4.5: Java Flow Allocator Interface.

(2) Flow Allocator Implementation

An application process talks to other application processes using the communication service provided by underlying IPC processes, and each IPC process registers the mapping between the application process above and itself within the DIF. This mapping information is currently flooded inside the DIF, and stored in directory forwarding table in each IPC process's RIB.

When the Flow Allocator (FA) receives a flow request through the FA interface, it first checks if the request is well informed. If not, it returns an error to the caller. Otherwise, the FA creates a Flow Allocator Instance (FAI) and passes the flow request to that instance. FAI is responsible for maintaining the flow during its lifetime. When a FAI is created with a flow request as input, the FA consults its directory forwarding table to find out which IPC process is bound to the destination application process. If such IPC process is found, FAI assigns a local port ID bound to the application, then sends an M_CREATE message containing the flow object to that IPC process's Management AE.

When the destination IPC process's Management AE receives the M_CREATE message containing a flow object, the FA interface is called and a new FAI is created with the request as an input. FAI checks if the destination application process is accessible or not, based on the access control policy. If so, FAI returns an M_CREATE_R containing the flow creation request result to the source IPC process. If the flow creation result is positive, the M_CREATE_R message also contains the remote port ID bound for the destination application process. Then the flow is created between the two application processes.

For each flow request, FA returns a dynamically assigned port ID to the requester, and the port ID is used to send and receive DTP messages. Also for each flow, FA maintains a flow message queue (defined in `rina/util/MessageQueue`) used for RMT purposes. When the **deallocateFlow** method is called, the flow is removed as well as its corresponding FAI.

4.2.7 Routing Daemon Package

The Routing Daemon (defined in `rina.routing/RoutingDaemon.java`) is responsible for the routing inside the DIF, and it is initiated by the Management AE after the IPC process successfully joins a DIF.

(1) Implementation

In the current version, we support a link state routing protocol. However, we have policy holders where new routing protocols can be implemented (see Section 6.3 for details). The Routing Daemon reads its routing policy settings (explained in Section 5.2.4) from the configuration file of the IPC process. The link state routing is implemented based on a Pub/Sub system provided by the RIB Daemon (Section 4.2.3). Once joining the DIF, the Routing Daemon of each IPC process creates three **PUB** events as follows.

```
SubscriptionEvent event = new SubscriptionEvent (EventType.PUB,
  checkNeighborPeriod ,"checkNeighborAlive");
this.ribDaemon.createEvent(event);

SubscriptionEvent event1 = new SubscriptionEvent (EventType.PUB,
  routingEntrySubUpdatePeriod ,"linkStateRoutingEntry");
this.ribDaemon.createEvent(event1);

SubscriptionEvent event2 = new SubscriptionEvent (EventType.PUB,
  routingEntrySubUpdatePeriod ,"linkStateRoutingEntryNeighborsReceived");
this.ribDaemon.createEvent(event2);
```

Also, for all its direct neighbors, each IPC process's routing daemon creates three corresponding **SUB** events.

```
SubscriptionEvent event = new SubscriptionEvent  (EventType.SUB,
  checkNeighborPeriod , "checkNeighborAlive", neighbor);
this.ribDaemon.createEvent(event);

SubscriptionEvent event1 = new SubscriptionEvent (EventType.SUB,
  routingEntrySubUpdatePeriod , "linkStateRoutingEntry", neighbor);
this.ribDaemon.createEvent(event1);

SubscriptionEvent event2 = new SubscriptionEvent (EventType.SUB,
  routingEntrySubUpdatePeriod , "linkStateRoutingEntryNeighborsReceived", neighbor);
this.ribDaemon.createEvent(event2);
```

(2) Pub/Sub Events details

In this section, we explain how the routing information is collected through the above three kinds of Pub/Sub events.

- (A) "checkNeighborAlive" event: when neighbors receive this event update from an IPC process , they mark this IPC process as "alive". For each neighbor, an IPC process attaches a timer (`rina.routing.util/CheckNeighborTimerTask`), and if it does not receive "checkNeighborAlive" update for a certain period of time, it marks the neighbor as "dead" and removes it from its direct neighbor list.
- (B) "linkStateRoutingEntry" event: the IPC process keeps sending its adjacent links' cost to all its neighbors, so that neighbors know its link cost information.
- (C) "linkStateRoutingEntryNeighborsReceived" event: the IPC process keeps sending all its link cost information received from a neighbor to its other neighbors. So the link cost information of each IPC process is flooded in the whole DIF.

Through events (B) and (C), once the Routing Daemon collects the link cost information of the whole DIF, it can build the forwarding table using Dijkstra's algorithm.

4.3 RINA Message Package

4.3.1 CDAP Message

The Common Distributed Application Protocol (CDAP) is used by RINA applications to exchange any structured application-specific data. A CDAP message generator is defined in `rina.message/CDAPMessageGenerator.java` to generate CDAP messages which are serialized by Google Protocol Buffers (GPB) [35].

4.3.2 EFCP Message

The Error and Flow Control Protocol (EFCP) includes a Data Transfer Protocol (DTP) and a Data Transfer Control Protocol (DTCP). In the current version, we only implement DTP, which is used by IPC processes to exchange Protocol Data Units (PDUs) which carry Service Data Units (SDUs).

The DTP implementation and its message generator are defined in `rina.message/DTP.java`. DTP is able to take a DTP message or a CDAP message as its payload. The DTP message is serialized using little endian.

4.4 IDD Package

The IDD (Inter-DIF Directory) is responsible for DIF name and application name resolution. It takes requests for the IDD Record of a DIF name or application name, finds the record and returns it. For a DIF name, the IDD record contains information of IPC processes that are responsible for enrolling new members into the DIF. For an application name, the IDD record contains information of the DIFs and their IPC processes which are able to help reach the application process. IDD can be implemented either in a distributed way or centralized way. In the current version, we implement IDD in a centralized way, and there is only one IDD process that is accessible by all (IPC or application) processes.

4.4.1 IDD Implementation

In our current version, the IDD process is reachable by any DIF-0 IPC process through the TCP shim layer (Section 4.6). The IDD process (defined in `rina.idd/IDDProcess.java`) keeps listening to a well-known TCP port, and for each client it creates an IDD Handler thread (defined in `rina.idd/IDDHandler.java`) to handle all the messages received from that client. Non DIF-0 IPC processes and application processes communicate with the IDD process using any of their underlying DIF-0 processes, namely each DIF-0 IPC process provides an IDD communication interface for higher level IPC processes and application processes. The IDD process has a database which stores information about registered DIFs and applications, so that it can serve query requests. The information is stored as an IDD record in the database, and the IDD Record is defined in `rina.object.internal/IDDRecord.java`.

4.4.2 IDD Message

There are three types of IDD messages: (1) Request, which contains the IDD query; (2) Response, which contains the returned IDD record; (3) Register, which contains the registration information about a DIF or an application.

A $Request$ message: either contains the **applicationNameInfo** field or the **difName** field, corresponding to an application query or a DIF query. A $Response$ message: could be a reply to a DIF name query and in this case it contains the authenticatorNameInfo object which includes a list of IPC processes that can enroll new members into the DIF; or it could be a reply to an application query and in this case it contains the iddResponse object which includes a list of entries containing the DIF name, IPC process information and names of supporting DIFs, which will help reach the application requested. A $Register$ message: either to register an application or a DIF. When the IDD receives a $Register$ message, it checks if the registration information is well formed and if so, it stores it in the IDD database.

4.5 RINA Object Package

This package contains all objects used in the current implementation. Objects carried in CDAP messages exchanged between processes are serialized using Google Protocol Buffers (GPB) [35]. The GPB proto files for all objects are in the appendix, where each proto file contains the details of each object.

4.5.1 Flow Object

In the current implementation, attributes used in the flow object include: source application naming information, source port ID, source address, source connection end-point ID, destination application naming information, destination port ID, destination address, destination connection end-point ID. Other attributes, such as quality of service and access control, are not supported in the current implementation.

4.5.2 Application Naming Information Object

The application naming information object includes four attributes: application process name, application process instance, application entity name and application entity instance.

4.5.3 Subscription Event Object

In the current implementation, attributes used in the subscription event object includes: event type (**Sub/Pub**), subscription ID, subscriber member list, update period, attributes list and attributes values. The `SubscriptionEvent` object has a *tolerance* field which is a relational-expression, and can be used to define more sophisticated subscription events. But in our current implementation, we do not support this *tolerance* field, and we will add this support later.

4.5.4 Other Objects

Other objects include: enrollment information object, directory forwarding entry object, DIF member information object, DIF neighbor list object, link-state routing entry object and IDD message object.

4.6 TCP Shim Layer Package

The TCP shim layer package is responsible for emulating physical connectivity using TCP connections for the level-0 DIFs. Each DIF-0 IPC process listens to one well-known TCP port, and two types of flows (Management and Data Transfer) between each pair of IPC processes are mapped on one single TCP connection. The TCP Flow Manager (defined in `rina.tcp/TCPFlowManager.java`) maintains all the TCP connections (emulated wires) between the DIF-0 IPC processes, and it is initiated by the IRM when a DIF-0 IPC process is created.

4.6.1 Implementation

In this version, there is only one TCP connection (emulated wire) between each pair of DIF-0 IPC processes. The TCP Flow Manager initiates all wires based on the settings in the IPC process's configuration file (Section 5.2.1) when the IPC process is created. For each IPC process, there is a thread (defined in `rina.tcp.util/IncomingCom.java`) listening to the well-known port. When another IPC process asks for a TCP connection through this port, an incoming communication handler (defined in `rina.tcp.util/IncomingComHandler.java`) is forked, which then creates a wire listener (defined in `rina.tcp.util/WireListenerTCP.java`) for each TCP connection. The wire listener is responsible for setting up the management flow or the data transfer flow upon request from the IPC process, and it is also responsible for multiplexing DIF-0 flows (Management or Data Transfer) onto the single wire. As TCP connections are bidirectional, for each outgoing TCP connection, a wire listener is also created to serve flow requests for DIF-0 flows.

4.6.2 DNS Process

We implement our own DNS to resolve a named interface to an IP address. The DNS process continuously listens to a well-known TCP port, and it is reachable by any DIF-0 IPC process through the TCP shim layer. When a DIF-0 IPC process is created, it registers the mapping between its name and IP address to the DNS process, so that any DIF-0 IPC process in the emulated physical network is reachable through a TCP connection. For each client, the DNS process (defined in `rina.dns/DNSProcess`) creates a DNS handler (defined in `rina.dns/DNSHandler`) that handles the DNS messages (*Register, Request and Response*) received. A DNS message generator is defined in `rina.dns/DNSMessage.java` to generate DNS messages which are serialized using GPB.

4.6.3 TCP Delimiting

Before a message is sent over the TCP connection, it is prepended with a variable-length integer (as `varint` defined by Google Protocol Buffers [35]) indicating the length of the message. The format of the variable-length integer is as follows, where the last byte is required, and other bytes are optional.

[<one-bit value 1><7-bit unsigned integer>] <one-bit value 0><7-bit unsigned integer>.

For each byte, the high-order bits indicates whether there are further bytes to come, and the low-order 7 bits are used to store the representation of the number in groups of 7 bits.

4.7 DAP Package

The DAP package includes the implementation of a Distributed Application Process (DAP) and its basic components. It can be extended by users to write their own application.

4.7.1 DAP component

Every Distributed Application Process (DAP) consists of the following components:
(1) **Resource Information Base (RIB)**: this component is responsible for storing all information useful for the management of a DAF (Section 4.2.2).

(2) **RIB Daemon**: this component is responsible for managing the information stored in the RIB and its veracity, updating and making states available to application processes (Section 4.2.3).

(3) **IPC Resource Manager (IRM)**: this component is responsible for managing the use of underlying IPC processes to allocate and maintain the connections with its peer Distributed Application Processes. It has the same function as that of the IRM of a non DIF-0 IPC process, as an IPC process can also be seen as an application process. This components is defined in `application.component.impl/IPCManagerImpl`.

(4) **Management Application Entity (Management AE)**: this component is responsible for handling all CDAP messages received from other application processes' management AE. Such messages include the CDAP messages used by IRM to set up an application connection, or CDAP messages used during dynamic DIF formation (Section 7.2). This components is defined in `application.ae/DAPManagementAE`.

4.7.2 Application-Specific Message Hanlder

Each application process has an application-specific message handler (defined in `application.component.util/RegularHandler.java`). It handles application messages exchanged between applications, and determines what to do based on the function the application performs.

4.8 RINA Node Package

In our RINA implementation, a *RINA node* (defined in `node/Node.java`) is a host where application processes and IPC processes reside. Based on the RINA node configuration file (Section 5.1), application processes and IPC processes are initialized by the RINA node. Each RINA node has a network management system (NMS), which manages the use of IPC processes on the RINA node.

4.9 Configuration Package

Each RINA node, application process and IPC process have their own configuration file. A *RINA configuration* object (defined in `rina.config/RINAConfig.java`) is used to read settings from the configuration file when initializing a

RINA node, application process or IPC process. Also the configuration object allows for dynamic configuration during the initiation of higher level IPC processes, instead of reading from a file.

The Apache log4j library [36] is used to keep track of the running status of the program. The logging property file (`log4j.properties`) specifies how program running information (such as info, debug, error) should be outputted, whether to the log file or to the console.

Chapter 5

Using RINA

In this chapter we explain how users can edit configuration files and use the RINA APIs to write their own networking applications. In the end, we provide a video streaming application as an example.

5.1 RINA Node Configuration

In our RINA implementation, a *RINA node* is a host where application processes and IPC processes reside. An application process talks with other application processes using the communication service provided by the IPC processes on the same RINA node. When running an application, users need to write the configuration file for each RINA node, specifying the application information and also information of the network management system which manages the use of IPC processes that may belong to different DIFs.

5.1.1 Network Management System Information

Each RINA node has a network management system, which manages the use of all IPC processes on the node. When a RINA node is initialized, all IPC processes on the node are also bootstrapped based on their configuration (Section 5.2). The network management system information includes the properties of each IPC process, such as DIF name, IPC process name and instance, and the IPC process's configuration file. As an example of network management system information, consider the following configuration: the RINA node has two IPC processes (BostonU1 and BostonU2), belonging to DIF1 and DIF2, respectively.

```
#first IPC process properties
IPC.1.DIF = DIF1
IPC.1.name = BostonU
IPC.1.instance = 1
IPC.1.configurationFile = ipcBosonU1.properties

#second IPC process properties
IPC.2.DIF = DIF2
IPC.2.name = BostonU
IPC.2.instance = 2
IPC.2.configurationFile = ipcBostonU2.properties
```

5.1.2 Application Information

Application information includes application name, application instance (optional), service name (optional), and information (IPC name and IPC instance) of underlying IPC processes that this application process uses to communicate with other application processes. Application name and instance together identify an application process. When an application is initialized, it registers the service it provides to the Inter-DIF Directory (IDD) service so that its service can be reached. Users can also write a separate configuration file for an application process if needed, and use the *RINA configuration* object (Section 4.9) to read the settings from the configuration file.

For example, as shown below, the application process has application name *myApp*, application instance *1*, and it provides video streaming service. Also it uses two IPC processes (*BostonU1 and BostonU2*) as its underlying IPC processes.

```
application.name = myApp
application.instance = 1
service.name = videoStreaming

#underlying IPC process information
#key is the concatenation of IPC Name + IPC Instance
underlyingIPC.1 = BostonU1
underlyingIPC.2 = BostonU2
```

Each underlying IPC process registers within the DIF the mapping between itself and the application process using it, such that when an IPC process's Flow Allocator (FA) is called to allocate a connection between applications, the FA can resolve the destination application to an IPC process in the DIF.

5.2 IPC Process Configuration File

Each IPC process has a configuration file. The configuration file specifies the TCP shim layer connectivity (for DIF-0 IPC process) or underlying DIFs information (for non DIF-0 IPC processes). Also the configuration file specifies IPC process's properties, such IPC name and IPC instance (which uniquely identify an IPC process inside a DIF), IPC DIF level, routing policy, and authentication policy.

5.2.1 TCP Shim Layer Setup

The configuration file for the DIF-0 IPC process specifies the TCP shim layer connectivity, namely, which IPC processes have a TCP connection (emulated wire) to this IPC process. As mentioned in the TCP shim layer section (Section 4.6), the TCP Flow Manager is responsible for emulating wires. The following properties are part of the configuration file related to the TCP shim layer:

```
#TCP Shim Layer Setup
neighbour.1 = BostonU2
neighbour.2 = BostonU3
neighbour.3 = BostonU4
```

The IPC process with the above properties has three neighbors (*BostonU2, BostonU3 and BostonU4*) , and the value of each property key is the concatenation of IPC process and IPC instance, for example, the first neighbor's IPC name is *BostonU*, and its IPC instance is *2*. If we want to add another IPC process as a neighbor, *e.g.*, (*Name: BostonU, Instance: 5*), we merely need to add one line as follows:

```
neighbour.4 = BostonU5
```

The IPC configuration file contains also the local TCP port used to listen for incoming connections to set up wires and the DNS information (including DNS name and its port) as follows:

```
# local TCP port
TCPPort = 11112  //this can be any valid port number

# DNS properties
rina.dns.name = localhost //this can be any reachable IP address
rina.dns.port = 11111 //this can be any valid port number
```

When a DIF-0 IPC process is created, it registers the mapping between its name (concatenation of IPC name and IPC instance), IP address as well local TCP port to DNS so that other IPC process can reach it and set up an emulated wire. Here the value of `rina.dns.name` can be an IP address or URL of the machine that hosts the DNS process.

As mentioned in the IDD section (Section 4.4), each DIF-0 IPC process communicates with the IDD process through the TCP shim layer. The IDD information (IP address or URL and TCP port) is also specified in each DIF-0 IPC process configuration file as follows:

```
#RINA IDD info
rina.idd.name = locahost  //this can be any reachable IP address
rina.idd.port = 8888 //this can be any valid port number
```

5.2.2 Underlying IPC Processes Properties

As a non DIF-0 IPC process can be seen as an application that only provides communication services, so it talks to other IPC processes in the same DIF also using communication service provided by lower level DIF IPC processes. So in its configuration file, we need to specify which underlying IPC processes it uses. This is the same as the configuration of underlying IPC processes for an application process as in Section 5.1.2.

5.2.3 IPC Process Properties

The configuration file also specifies IPC naming information, including IPC name and IPC instance, as well as the DIF level as shown below. The IPC process's name and instance are *BostonU* and *1*, and it is a DIF-0 IPC process.

```
rina.ipc.name = BostonU
rina.ipc.instance = 1
rina.ipc.level = 0
```

An IPC process can be started as a member of a given DIF. To bootstrap this case, an IPC process needs to be initiated as the first member in the DIF with the following lines, which define the IPC process as enrolled in a DIF with DIF name *myDIF*. The address of this IPC process will be initialized based on the internal address policy (Section 5.2.4).

```
rina.dif.enrolled = true
rina.dif.name = myDIF
```

If an IPC process is instead initialized not as a part of any DIF, but the user plans to bootstrap its enrollment into a specific DIF *myDIF*, the IPC process has to wait to join such DIF, and hence its configuration file has to contain the following lines. The IPC process's internal RINA address will be assigned by the enroller based on the internal address policy (Section 5.2.4).

```
rina.dif.enrolled = false
rina.dif.name = myDIF
```

The IPC process gets the enroller's information from IDD by sending an IDD query with DIF name. Also we can also manually set the enroller's information for an IPC process, and IPC process don't have to query IDD. As shown below, the IPC process will be enrolled by IPC process whose name is *BostonU* and instance is 2.

```
rina.authenticator.apName = BostonU
rina.authenticator.apInstance = 2
```

For all DIF-0 IPC processes, we manually set its underlying DIF information (TCP shim layer configuration) as follows:

```
rina.underlyingDIF.name.1 = wire1
rina.underlyingDIF.name.2 = wire2
```

This information is used by the enroller during the enrollment procedure to determine which of the IPC processes that already joined the DIF is the new member's neighbor: if two IPC processes have a same underlying DIF, then these two IPC processes are direct neighbors. This underlying DIF information is mainly used by the routing daemon of the IPC process to collect routing information (Section 4.2.7).

5.2.4 IPC Process Policies

In this section, we explain the configuration setting of different policies for the IPC processes. The policies supported with this implementation version include, authentication, internal addressing policy, and routing policy.

(1) Authentication Policy

During the enrollment procedure, some authentication information is optionally required by the enroller IPC process. The following lines in the configuration file specify that the authentication needs username and password.

```
rina.enrollment.authenPolicy = AUTH_PASSWD
rina.ipc.userName = BUuser
rina.ipc.passWord = BUpwd
```

After joining the DIF, an IPC process gets the authentication policy from the enroller, and uses such assigned policy to enroll new members in the future. Other authentication polices might include, for example, RSA or SSH. Our current implementation only supports username and password, or no authentication. The setting for no authentication can be specified as follows:

```
rina.enrollment.authenPolicy = AUTH_NONE
```

(2) Internal Addressing Policy

When an IPC process is the first member of a certain DIF, it is bootstrapped with an internal private address assigned by itself. When an IPC process enrolls a new member into the DIF, it assigns an internal RINA address to the new member. The addressing policy specifies how the address is generated. Each private address can be randomly generated, and if the address generated is not used then it is assigned to the new member, otherwise another address is generated. The address might follow a certain scheme. The following is an example of using the random addressing policy.

```
rina.address.policy = RANDOM
```

For simplicity and also to make the experiment repeatable, users can also manually set the internal address for IPC processes in their configuration files as long as there is no conflict.

(3) Routing policy

The Routing Daemon (Section 4.2.7) is responsible for routing inside the DIF. For example, as shown below, the IPC process uses link-state routing, it sends routing updates to its neighbors every 2 seconds, and checks whether its neighbor is alive every second. The link cost is calculated using hop count.

```
#routing related properties
rina.routing.protocol = linkState //routing policy
rina.routingEntrySubUpdatePeriod = 2 //unit: second
rina.checkNeighborPeriod = 1 //unit: second
rina.linkCost.policy = hop //counts
```

5.3 Writing Your Own Application

In this section, we explain how users can write their own applications using APIs provided in our RINA implementation, and also how to support more RIB Daemon Pub/Sub events in case applications need them.

5.3.1 RINA APIs

In this part, we explain three sets of RINA APIs that can be used to write an application.

(1) RIB Daemon API

The RIB Daemon API (Table 5.1) is used to retrieve information from the either local RIB or a remote application process's RIB.

RIB Daemon Interface
public int createEvent(SubscriptionEvent subscriptionEvent);
public void deleteEvent(int subscriptionID);
public Object readSub(int subID);
public void writePub(int pubID, byte[] obj);

Table 5.1: Java RIB Daemon Interface.

For example, if we want to create a `Pub` event with attribute name *attribute1*, and update period of *3* seconds, we first create a `SubscriptionEvent` object [1](defined in `rina.object.internal/SubscriptionEvent.java`) as follows:

```
SubscriptionEvent event = new SubscriptionEvent (EventType.PUB,3,"attribute1");
```

then use the **createEvent** API to create this `Pub` event as follows, which returns the event ID *pubID*:

```
int pubID = ribDaemon.createEvent(event);
```

To publish the content (in bytes), we use the **writePub** API along with the *pubID* as follows:

```
ribDaemon.writePub(pubID, object);
```

If another application process wants to subscribe to the `Pub` event above, and assuming the publisher application name is *app1*, we first create a `SubscriptionEvent` object as follows:

```
SubscriptionEvent event = new SubscriptionEvent (EventType.SUB,3,"attribute1");
```

then use the **createEvent** API to create this `Sub` event as follows, which returns the event ID *subID*:

```
int subID = ribDaemon.createEvent(event);
```

To retrieve information from this `Sub` event, we use the **readSub** API and store information published into an object as follows:

```
Object object = ribDaemon.readSub(subID)
```

The set of **Pub/Sub** events need to be predefined in the RIB Daemon so they it can be used. Details of defining a new set of **Pub/Sub** events are explained in Section 5.3.2.

(2) IRM API

The IRM API (Table 5.2) is used to create connections between application processes via underlying supporting DIFs.

IRM Interface
public int allocateFlow(Flow flow);
public void deallocateFlow(int handleID);
public void send(int handleID, byte[] msg) throws Exception;
public byte[] receive(int handleID);

Table 5.2: Java IRM Interface.

Let us assume that an application process (with application name: *app* and application instance: *1*) wants to create a connection to another application process (with application name: *app* and application instance: *2*). First we need to establish a flow, and we do so by creating a `Flow` object (defined in `rina.object.internal/Flow.java`) as follows:

```
Flow flow = new Flow("app", "1", "app", "2");
```

then we use the **allocateFlow** API that returns a *handleID* used to send and receive messages:

```
int handleID = irm.allocateFlow(flow);
```

To send a message with byte value *msgSend* through this connection, the **send** API is used as follows:

```
irm.send(handleID, msgSend);
```

To receive a message through this connection, the **receive** API is used as follows:

```
byte[] msgRecv = irm.receive(handleID);
```

[1]The `SubscriptionEvent` object has a *tolerance* field which is a relational-expression, and the *tolerance* field can define more sophisticated events, for example when the desired value is bigger than a certain threshold. But in our current implementation, we do not support this *tolerance* field, and we will add this support later.

To remove this connection, the **deallocateFlow** method is used as follows:

```
irm.deallocateFlow(handleID);
```

(3) Application Registration API

The application registration API (Table 5.3) is used by an application to register the service it provides to the Inter-DIF Directory (IDD) service so that it can be found by other application processes.

Application Registration Interface
public void registerServiceToIDD(String serviceName);
public void unregisterServiceFromIDD(String serviceName);

Table 5.3: Java Application Registration Interface.

Users can write their own application by extending the `Application` class defined in `application/Application.java`. For example, if an application process provides relay service to another application (with application name: *app* and application instance: *1*) , the **registerServiceToIDD** API is used as follows:

```
registerServiceToIDD("relayApp1");
```

If the application stops providing this service, the **unregisterServiceFromIDD** API is used as follows:

```
unregisterServiceFromIDD("relayApp1");
```

5.3.2 Defining New Pub/Sub Events

In this section, we explain how to define a new set of **Pub/Sub** event in the RIB Daemon. We assume that the new event's attribute name is *"newAttribute"*. In the package `rina.ribDaemon.util`, first modify the following method in the `Publisher.java` file,

```
private void updatePubValue()
```

and add the following code, specifying the byte array value that the publisher wants to publish to the subscribers in the new event:

```
else if (this.attributes.equals("newAttribute"))
{
   //newAttribute supporting code here
}
```

Next modify the following method in the `EventHandler.java` file,

```
public void updateSubEvent(byte[] value)
```

and add the following code, specifying how the **Sub** event value is updated when a subscriber receives an update from the publisher:

```
else if (this.attributes.equals("newAttribute"))
{
   //newAttribute supporting code here
}
```

5.4 RINA Video Streaming Application Example

In this section, we show how a video streaming application is set up using our RINA implementation.

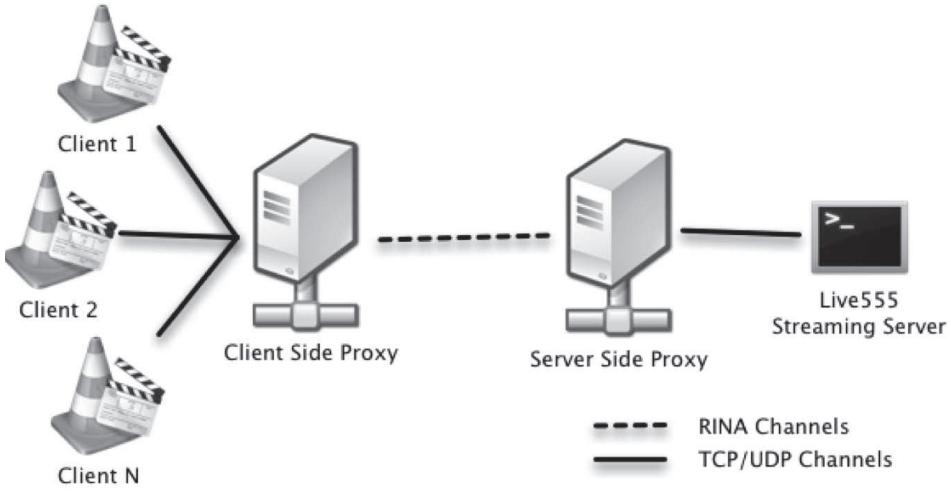

Figure 5.1: Video Streaming Application through proxies using RINA communication service.

5.4.1 RINA Video Streaming Application

RINA video streaming application is a streaming proxy that supports Real-Time Streaming Protocol (RTSP), and redirects all traffic between the media player client and the video streaming server to the communication service provided by the DIF. As shown in Figure 5.1, the main component of RINA video streaming application consists of a client side proxy and a server side proxy which are connected via a RINA connection. Clients 1 to N are media player clients that issue RTSP requests to the client side proxy. The requests go through the underlying channels provided by RINA, and arrive at the server side proxy, which issues the requests to the actual streaming server on behalf of clients. On each client we run a VLC media player [38], and the server uses Live555 Streaming Media Server [39] to stream videos.

In out prototype, we include video streaming related java files in the `video` package. The server side proxy is defined in `video.serverProxy/ServerProxy.java`, and the client side proxy is defined in `video.clientProxy/ClientProxy.java`. The details of the video streaming application can be found at [37].

5.4.2 RINA Streaming in Action

(1) IPC Process Configuration

The DIF topology underlying the streaming application is shown in Figure 5.2, and consists of five IPC processes. Client side proxy is using *IPC1* as underlying IPC process, and sever side proxy is using *IPC3* as underlying IPC process.

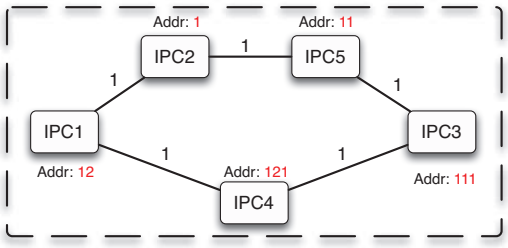

Figure 5.2: DIF topology.

In each IPC process's configuration file, the DNS and IDD information, routing and enrollment policies (including authentication policy and address assignment policy) are as follows. The simple address assignment policy `PATTERN1` we use here

works as follows: assume the enroller's address is *n*, and the new member is the m^{th} enrolled member by the enroller, then the new member's address is *10*×*n* + *m* where $m < 10$.

```
rina.dns.name = csa2.bu.edu
rina.dns.port = 11111

rina.idd.name = csa2.bu.edu
rina.idd.port = 8888

rina.routing.protocol = linkState
rina.routingEntrySubUpdatePeriod = 2
rina.checkNeighborPeriod = 2
rina.linkCost.policy = hop

rina.enrollment.authenPolicy = AUTH_PASSWD

rina.addressPolicy = PATTERN1
```

IPC2 is the first member in *DIF1*, so it is able to enroll new members from the beginning and its internal private address is set to be 1 by default. *IPC1*, *IPC3*, *IPC4* and *IPC5* then join *DIF1* (see Section 5.4.3 for details of how each IPC process is enrolled). Next we show the settings for each IPC process, and the settings explained above for DNS, IDD, routing and enrollment polices are not repeated here.

(A) Configuration for *IPC2*:

```
rina.ipc.level = 0
rina.ipc.name = BostonU
rina.ipc.instance = 2

rina.dif.enrolled = true
rina.address = 1 // IPC2 is the first member in the DIF, here we manually set IPC2's address to 1
rina.dif.name = DIF1
rina.ipc.userName = BU
rina.ipc.passWord = BU

rina.underlyingDIF.name.1 = wire1
rina.underlyingDIF.name.2 = wire3

TCPPort = 11113

neighbour.1 = BostonU1
neighbour.2 = BostonU5
```

(B) Configuration for *IPC1*:

```
rina.ipc.level = 0
rina.ipc.name = BostonU
rina.ipc.instance = 1

rina.dif.enrolled = false
rina.authenticator.apName = BostonU
rina.authenticator.apInstance = 2

rina.dif.name = DIF1
rina.ipc.userName = BU
rina.ipc.passWord = BU

rina.underlyingDIF.name.1 = wire1
rina.underlyingDIF.name.2 = wire2

TCPPort = 11112

neighbour.1 = BostonU2
neighbour.2 = BostonU4
```

(C) Configuration for *IPC3*:

```
rina.ipc.level = 0
```

```
rina.ipc.name = BostonU
rina.ipc.instance = 3

rina.dif.enrolled = false
rina.authenticator.apName = BostonU
rina.authenticator.apInstance = 5

rina.dif.name = DIF1
rina.ipc.userName = BU
rina.ipc.passWord = BU

rina.underlyingDIF.name.1 = wire4
rina.underlyingDIF.name.2 = wire5

TCPPort = 11114

neighbour.1 = BostonU4
neighbour.2 = BostonU5
```

(D) Configuration for *IPC4*:

```
rina.ipc.level = 0
rina.ipc.name = BostonU
rina.ipc.instance = 4

rina.dif.enrolled = false
rina.authenticator.apName = BostonU
rina.authenticator.apInstance = 1

rina.dif.name = DIF1
rina.ipc.userName = BU
rina.ipc.passWord = BU

rina.underlyingDIF.name.1 = wire2
rina.underlyingDIF.name.2 = wire5

TCPPort = 11115

neighbour.1 = BostonU1
neighbour.2 = BostonU3
```

(E) Configuration for *IPC5*:

```
rina.ipc.level = 0
rina.ipc.name = BostonU
rina.ipc.instance = 5

rina.dif.enrolled = false
rina.authenticator.apName = BostonU
rina.authenticator.apInstance = 2

rina.dif.name = DIF1
rina.ipc.userName = BU
rina.ipc.passWord = BU

rina.underlyingDIF.name.1 = wire3
rina.underlyingDIF.name.2 = wire4

TCPPort = 11116

neighbour.1 = BostonU2
neighbour.2 = BostonU3
```

(2) RINA Nodes Configuration

As shown in Figure 5.3, the client side proxy uses *IPC1* as its underlying IPC process, and the server side proxy uses *IPC3* as its underlying IPC process. The configuration file for RINA node where the client side proxy resides is as follows, where

`ipcBostonU1.properties` is the configuration file for *IPC1*.

```
#Network management system information
IPC.1.DIF = DIF1
IPC.1.name = BostonU
IPC.1.instance = 1
IPC.1.configurationFile = ipcBostonU1.properties

#Client side proxy application process information
application.name = clientProxy
underlyingIPC.1 = BostonU1
```

The configuration file for RINA node where the server side proxy resides is as follows, where `ipcBostonU3.properties` is the configuration file for *IPC3*.

```
#Network management system information
IPC.1.DIF = DIF1
IPC.1.name = BostonU
IPC.1.instance = 3
IPC.1.configurationFile = ipcBostonU3.properties

#Server side proxy application process information
application.name = serverProxy
underlyingIPC.1 = BostonU3
```

There is no application process on the other three RINA nodes where *IPC2, IPC4* and *IPC5* reside. The configuration files for RINA node where IPC2 resides is as follows, where `ipcBostonU2.properties` is the configuration file for *IPC2*.

```
#Network management system information
IPC.1.DIF = DIF1
IPC.1.name = BostonU
IPC.1.instance = 2
IPC.1.configurationFile = ipcBostonU2.properties
```

The configuration file for RINA node where IPC4 resides is as follows, where `ipcBostonU4.properties` is the configuration file for *IPC4*.

```
#Network management system information
IPC.1.DIF = DIF1
IPC.1.name = BostonU
IPC.1.instance = 4
IPC.1.configurationFile = ipcBostonU4.properties
```

The configuration file for RINA node where IPC5 resides is as follows, where `ipcBostonU5.properties` is the configuration file for *IPC5*.

```
#Network management system information
IPC.1.DIF = DIF1
IPC.1.name = BostonU
IPC.1.instance = 5
IPC.1.configurationFile = ipcBostonU5.properties
```

Each RINA node is boostrapped based on the its configuration file. Once the RINA connection between the client proxy and server proxy is set up, the client proxy is ready to accept video requests from a media player client for the streaming server.

5.4.3 Video Streaming Demonstration

In this walk-through demonstration, we first start the DNS process and the IDD process. Then start the *IPC2* process which is the first member in the DIF with address 1. Next we start the *IPC5* process and which joins the DIF and get assigned address 11 by *IPC2* . Then we start the client proxy process which creates the *IPC1* process as part of it, so *IPC1* joins the DIF and get assigned address 12 by *IPC2* and then client proxy uses *IPC1* as its underlying IPC. Next we start *IPC4*, which joins the DIF and get assigned address 121 by *IPC1*. Last we start theserver proxy process which creates the *IPC3* process as part of it, so *IPC3* joins the DIF and get assigned address 111 by *IPC5*, then the server proxy uses *IPC3* as its underlying IPC. When all processes in the DIF are started, after few seconds the Routing Daemon of each IPC process computes the routing table.

Then the client side proxy creates a RINA connection to the server side proxy using the IRM API's **allocateFlow** API. The connection is mapped to the path *IPC1-IPC4-IPC3* within the underlying DIF, since this is the shortest path between *IPC1* and *IPC3* as shown in Figure 5.3.

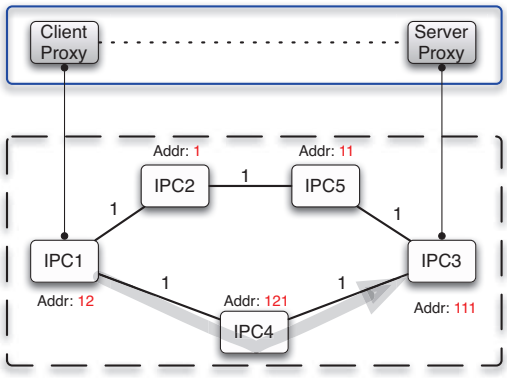

Figure 5.3: Connection between client proxy and server proxy is through *IPC4*.

Then we let *IPC4* go down by stopping the *IPC4* process and on the media player side, we can see the video stops for a moment. The Routing Daemon of each IPC process detects the topology change, and later converges to a new routing table after few seconds, so the connection now is mapped to another path *IPC1-IPC2-IPC5-IPC3* as shown in Figure 5.4. Then on the media player side, the video resumes playing.

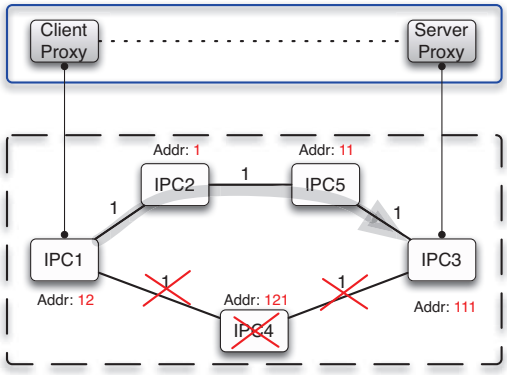

Figure 5.4: *IPC4* is down, then connection between client proxy and server proxy is through *IPC2* and *IPC5*.

Then we let *IPC4* returns to the DIF by starting the *IPC4* process again. The connection is then re-mapped to the shortest path *IPC1-IPC4-IPC3* as shown in Figure 5.5. When rejoining the DIF, *IPC4* is assigned a new address 122 instead of the old address 121 as it is treated as a new member.

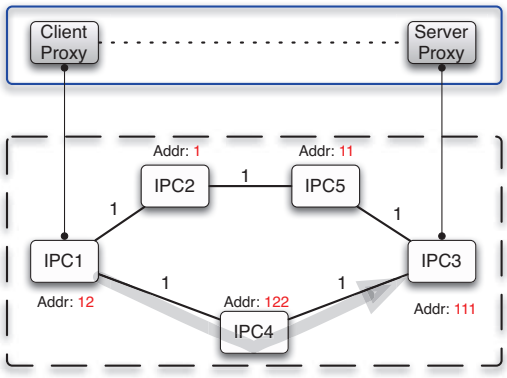

Figure 5.5: *IPC4* is back, then connection between client proxy and server proxy is through *IPC4* again.

Chapter 6

Programming RINA

In this chapter, we explain how programmers can program with RINA and write their own policies using the policy holders in our implementation.

6.1 Authentication Policy

Currently, we support two kinds of authentication policies: (1) no authentication, and (2) using user name and password. During the enrollment procedure, an $M\_CONNECT$ is sent by the IPC process that wants to join the DIF, and it contains the authentication information, so to add a new authentication policy, we need to modify both sender's side and receiver's side of this $M\_CONNECT$ message.

On the sender's side (defined in `rina.application.entity/ManagementAE.java`), modify the **enrollment** method by setting the *authTypes_t* field in the $M\_CONNECT$ to the new policy and letting the *authValue_t* field carry the authentication object (see the proto file in the appendix for CDAP message for details) such as RSA keys. Currently, the *authTypes_t* field is an enum type which can be set to four values **AUTH_NONE, AUTH_PASSWD, AUTH_SSHRSA, AUTH_SSHDSA**. If the programmer wants to add another policy other than RSA and DSA, the proto file for CDAP message needs to be modified by adding another enum value for *authTypes_t*, then recompiled by GPB [35] to generate a new CDAP message java file, and replace `rina.message/CDAP.java`. On the receiver's side (defined in `rina.application.entity/ManagementAEHandler.java`), modify the **handle_M_CONNECT** method and add the corresponding code which updates the authentication result indicated by variable *result*.

Once the new authentication policy is added, if users want to use the new policy, they only need to specify in the IPC process's configuration file the authentication policy as follows:

```
rina.enrollment.authenPolicy = myNewAuthenPolicy
```

6.2 Internal Addressing Policy

When an IPC process is the first member of a certain DIF, it is bootstrapped with an internal private address assigned by itself. When a new IPC process joins the DIF, it is assigned an internal private RINA address by the enroller. IPC processes use the internal private address to communicate with other IPC processes in the same DIF. Assume the new addressing policy that the programmer wants to add is *myNewAddressPolicy*, the **generateRINAAddr** method in `rina.rib.impl/RIBImpl.java` needs to be modified by adding the corresponding code as follows:

```
if(addressPolicy.equals("RANDOM"))
{
  . . . . . .
}else if(addressPolicy.equals("DEFAULT"))
{
  . . . . . .
}else if(addressPolicy.equals("myNewAddressPolicy"))
{
  //Implement new policy here
}
```

Once the new policy is added, if users want to use the policy, they only need to specify in the IPC process's configuration file the addressing policy as follows:

```
rina.address.policy = myNewAddressPolicy
```

6.3 Routing Policy

6.3.1 Routing Protocol

In the current version, we support a link state routing protocol, but programmers can implement their own routing protocols. Assume the new routing policy that the programmer wants to add is "*myNewRoutingPolicy* ", the programmer needs to modify the constructor method in `rina.routing/RoutingDaemon.java` as follows:

```
//POLICY HOLDER
if(this.routingProtocol.equals("linkState"))
{
  this.linkStateRoutingInfo = new LinkStateRoutingInfo(this.rinaAddr, this.forwardingTable);
  this.rib.addAttribute("linkStateRoutingInfo", this.linkStateRoutingInfo);
}else if (this.routingProtocol.equals("myNewRoutingPolicy"))
{
  //Implement new policy here
}
```

What's more, programmers may need to update the RIB Daemon to add more Pub/Sub events (See Section 5.3.2 for details) which are needed for the new routing protocol. Once the new routing policy is added, if users want to use the new routing policy, they only need to specify in the IPC process's configuration file the routing policy as follows:

```
rina.routing.protocol = myNewRoutingPolicy
```

6.3.2 Link Cost

Currently our implementation supports only hop count as the link cost. New policies of determining the link cost can also be implemented for the link state routing. Link cost is determined when an IPC process receives an *linkStateRoutingEntry* event update from its neighbors (defined in `rina.ribDaemon.util/EventHandler.java`). Assume the new policy programmer wants to add is "*myLinkCostPolicy*". In the **updateSubEvent** method, modify the updating part for *linkStateRoutingEntry* as follows:

```
//POLICY HOLDER
if(this.linkCostPolicy.equals("hop")) // hop count is the cost
{
  cost =1;
}else if (this.linkCostPolicy.equals("myNewLinkCostPolicy"))
{
  // Implement new link cost policy here
}
```

Once the new link cost policy is added, if users want to use the new policy, they only need to specify in the configuration file the link cost policy as follows:

```
rina.linkCost.policy = myNewLinkCostPolicy
```

Chapter 7

Appendix A: Flow Allocation

In this section, we explain details of different scenarios during the flow allocation between two processes when an IRM **allocateFlow** call is invoked (Section 4.2.4).

7.1 Flow Allocation Over An Existing DIF

In this section, we explain how a flow is allocated between two (IPC or application) processes through an existing underlying DIF. Here we show two cases, in the first case the flow is a one-hop flow in the underlying DIF, and in the second case the flow is a multi-hop flow.

7.1.1 Case 1

AppA wants to allocate a flow to *AppB* through an IRM API call. When AppA's IRM receives the request, it looks up its RIB, and finds one of its underlying IPC process: *IPC1* is able to reach *AppB* through *IPC2* . Then through the IPC API call, *IPC1*'s flow allocator gets the request to allocate the flow between *AppA* and *AppB*.

In this case, *IPC1* and *IPC2* are neighbors. An M_CREATE message containing the flow request is sent from *IPC1*'s Management AE to *IPC2*'s Management AE over the existing Management flow, and the flow request includes the local port bound between *IPC1* and *AppA*. When *IPC2* receives the request, it asks if *AppB* agree to create the flow. If *AppB* agrees, an M_CREATE_R containing the local port bound between *IPC2* and *AppB* is returned from *IPC2* to *IPC1*'s Management AE. Then the flow is created between *AppA* and *AppB*.

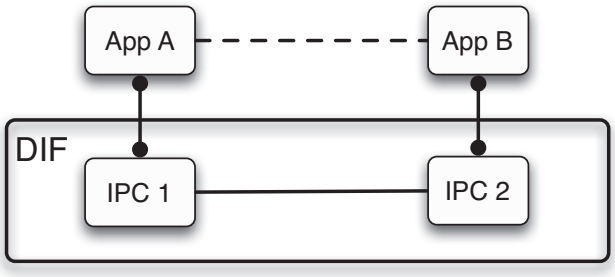

Figure 7.1: Flow created between direct neighbors IPC.

7.1.2 Case 2

AppA wants to allocate a flow to *AppB* through an IRM API call. When *AppA*'s IRM receives the request, it looks up its RIB, and finds one of its underlying IPC process: *iPC1* is able to reach *AppB* through *IPC3* . Then through the IPC API call, *IPC1*'s flow allocator gets the request to allocate the flow between *AppA* and *AppB*.

In this case, *IPC1* and *IPC3* are not neighbors, but since they are in the same DIF, so they are reachable to each other. *IPC1*'s Management AE will send a *M_CREATE* message to *IPC2*'s Management AE over the Management flow between them first, then *IPC2*'s Management AE will relay this CDAP message to *IPC3*'s Management AE over the Management flow between them. Similar to case 1, a M_CRATE_R message is sent from *IPC3* to *IPC1*'s Management AE, and the message is also relayed by *IPC2*'s Management AE.

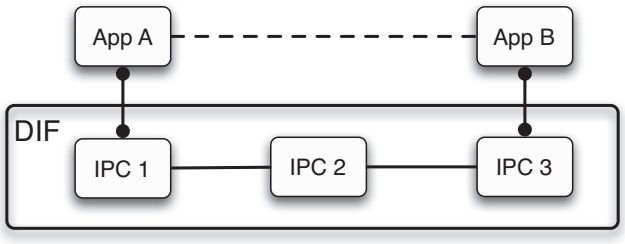

Figure 7.2: Flow created between non-neighbors.

7.2 Dynamic DIF Formation

When two (IPC or application) processes try to establish a communication flow, in last two examples, we show when there are existing common underlying DIF, however if they cannot find a common existing DIF to communicate over a new (higher level) DIF needs to be dynamically created.

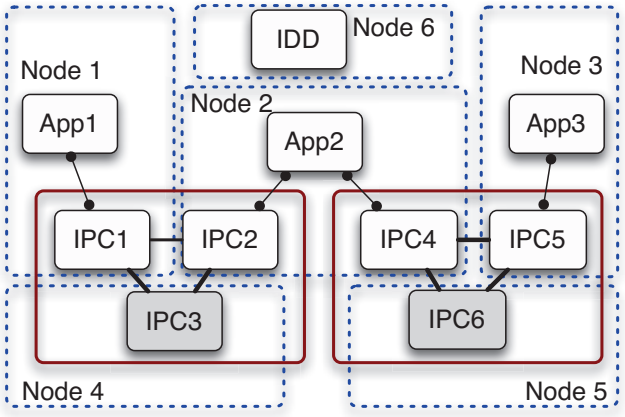

Figure 7.3: Dynamic DIF Formation: *App2* has underlying IPC processes in both *DIF1* and *DIF2*, and uses the Inter-DIF Directory (IDD) mechanism to register a relay service between them.

We consider three application processes that use two 0-level DIFs, *DIF1* and *DIF2* to communicate. Each underlying facility has three IPC processes. In particular, *App1* and *App3* use *IPC1* and *IPC5* as underlying IPCs for their communication in *DIF1* and *DIF2*, respectively, while *App2* uses *IPC2* and *IPC4* for its communications in *DIF1* and *DIF2* (Figure 7.3).

App1 wants to establish a flow with *App3*, and since it cannot find a common underlying DIF, it triggers the Inter-DIF Directory (IDD) service mechanism and discovers which DIF it needs to join to establish a flow with *App3*. Previously registered as relay service for *App3*, the address of *App2* is returned to *App1*. Then *App1* sends an M_CREATE message to *App2* indicating it wants to allocate a flow to *App3*. After receiving this message, *App2* will initiate new 1-level *DIF3* by creating *IPC8* which is the first member in *DIF3* and able to enroll new members. Then *App2* sends an M_CREATE to *App3* inviting *App3* to create a new IPC process to join the new created *DIF3*. After *IPC9* created by *App3* joins *DIF3*, *App3* sends an M_CREATE_R to *App2* telling that it is ready to accept flow creation request. Then *App2* sends an M_CREATE_R to

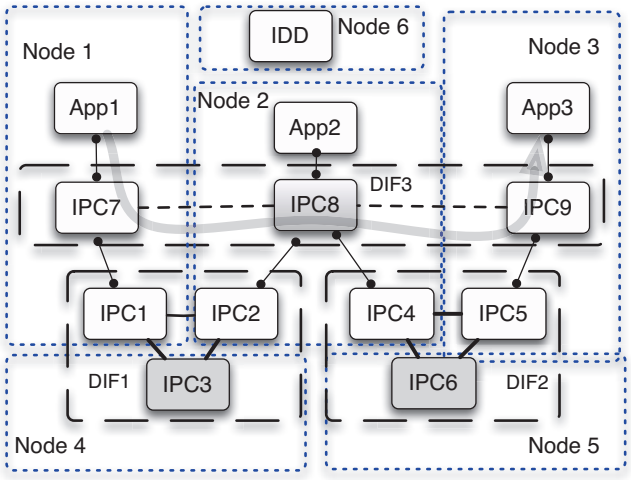

Figure 7.4: Dynamic DIF Formation: 1-level DIF3 is dynamically formed by `App2` upon `App1`'s request to establish a flow with `App3`.

App1 inviting *App1* to create a new IPC process to join the new created *DIF3*. Once *IPC7* is created by *App1*, *DIF3* is able to provide communication service between *App1* and *App3* (Figure 7.4). Then the flow between *App1* and *App3* is created similar to case 2 in section 7.1.

Chapter 8

Appendix B: Proto files

In order to use the Google Buffer Protocol serialization format, programmers need to specify how they want their information to be serialized. As extensively clarified in [35], each protocol buffer message is a small logical record of information, containing a series of name-value pairs. In this section, we list three proto files used in our implementation, and the rest protos files can be found in the prototype source code.

8.1 Flow Object

The following is the GPB proto file for the flow object.

```
package rina.object.gbp;
import "ApplicationProcessNamingInfoMessage.proto";
import "QoSSpecification.proto";
import "PropertyMessage.proto";

message connectionId_t
{  //information to identify a connection
        optional uint32 qosId = 1;
        //identifies the QoS cube under which this flow is classified
        optional uint32 sourceCEPId = 2;
        //identifies the source connection endpoint (unique within the source IPC process)
        optional uint32 destinationCEPId = 3;
        //identifies the destination connection endpoint (unique within the destination IPC process)
}

message flow_t
{  //Contains the information to setup a new flow

        required rina.messages.applicationProcessNamingInfo_t sourceNamingInfo = 1;
        //The naming information of the source application process
        required rina.messages.applicationProcessNamingInfo_t destinationNamingInfo = 2;
        //The naming information of the destination application process
        required uint64 sourcePortId = 3;
        //The port id allocated to this flow by the source IPC process
        optional uint64 destinationPortId = 4;
        //The port id allocated to this flow by the destination IPC process
        required uint64 sourceAddress = 5;
        //The address of the source IPC process for this flow
        optional uint64 destinationAddress = 6;
        //The address of the destination IPC process for this flow
        repeated connectionId_t connectionIds = 7;
        //The identifiers of all the connections associated to this flow
        optional uint32 currentConnectionIdIndex = 8;
        //Identifies the index of the current active connection in the flow
        optional uint32 state = 9;
        //?
        optional rina.messages.qosSpecification_t qosParameters = 10;
        //the QoS parameters specified by the application process that requested this flow
```

```
        repeated property_t policies = 11;
        //the set of policies selected by the IPC process
        repeated property_t policyParemeters = 12;
        //the set of parameters associated to the policies
        optional bytes accessControl = 13;
        // ?
        optional uint32 maxCreateFlowRetries = 14;
        //Maximum number of retries to create the flow before giving up
        optional uint32 createFlowRetries = 15;
        //The current number of retries
        optional uint32 hopCount = 16;
        //While the search rules that generate the forwarding table should allow for a natural
            termination condition, it seems wise to have the means to enforce termination
}
```

8.2 Subscription Event Object

```
package rina.object.gpb;
enum eventType_t
{
        SUB = 0;
        PUB = 1;
}

enum actionType_t
{
        NOTIFY = 0;
        RECORD = 1;
}

message subscriptionEvent_t
{  //Contains the information for a pub/sub event

        required eventType_t eventType = 1;
        repeated string attributeList = 2;
        optional int32 subscriptionID = 3;
        optional string relationExpression = 4;
        optional double tolerance = 5;
        repeated string memberList = 6;
        optional actionType_t actionType = 7;
        optional double updatePeriod = 8;
        optional bytes  value = 9;
}
```

8.3 IDD message

The IDD Message Object is serialized using GPB, and the proto file is defined as below.

```
package rina.object.gpb;

import "ApplicationProcessNamingInfoMessage.proto";

enum opCode_t {
        Request = 0;
        Response = 1;
        Register = 2;
}

message iddResponse_t{
        optional string difName = 1;
        optional rina.object.gpb.applicationProcessNamingInfo_t ipcProcessNameInfo = 2;
        repeated string supportingDIFNames = 3;
```

```
}

message iddMessage_t{
        required opCode_t opCode = 1;
        optional uint32 result = 2;
        optional rina.object.gpb.applicationProcessNamingInfo_t applicationNameInfo = 3;
        optional uint32 hopCount = 4;
        optional uint64 timeStamp = 5;
        repeated iddResponse_t iddResponse = 6;
        optional string difName = 7; // query for authenticator of a certain DIF
        repeated rina.object.gpb.applicationProcessNamingInfo_t authenticatorNameInfo = 8;
        // all IPC processes that can enroll new members
}
```

Bibliography

[1] A Technical History of CYCLADES, Technical Histories of the Internet and other Network Protocols (THINK), University of Texas, 11 June 2002.

[2] Xerox System Integration Standard - Internet Transport Protocols (Xerox, Stamford, 1981).

[3] James Martin, Joe Leben, DECnet Phase V: An OSI Implementation. Digital Press, 1992. ISBN 1-55580-769-0.

[4] ISO/IEC 7498-1:1994, Information technologyOpen Systems InterconnectionBasic Reference Model: The Basic Model.

[5] XIA: eXpressive Internet Architecture Project. http://www.cs.cmu.edu/∼xia/.

[6] MobilityFirst Future Internet Architecture Project. http://mobilityfirst.winlab.rutgers.edu/.

[7] Named Data Networking Project. http://named-data.net/.

[8] NEBULA Project. http://nebula-fia.org/.

[9] J. Day, I. Matta and K. Mattar. Networking is IPC: A Guiding Principle to a Better Internet. In Proc. of ReArch 2008. Madrid, Spain.

[10] The RINA specification handbook. Pouzin Society. January 2013.

[11] J. Day. Patterns in Network Architecture: A Return to Fundamentals. Prentice Hall, 2008.

[12] Boston University RINA lab website. http://csr.bu.edu/rina.

[13] The IRATI Project. http://irati.eu/.

[14] TRIA Network Systems, LLC. http://www.trianetworksystems.com/.

[15] Y. Wang, F. Esposito, and I. Matta. Demonstrating RINA using the GENI Testbed. The Second GENI Research and Educational Experiment Workshop (GREE 2013), Salt Lake City, UT, March 2013.

[16] ns-2. http://www.isi.edu/nsnam/ns/.

[17] ns-3. http://www.nsnam.org/.

[18] ns3 manual. http://www.nsnam.org/docs/release/3.17/manual/ns-3-manual.pdf/.

[19] ns-3 model library. http://www.nsnam.org/docs/release/3.17/models/ns-3-model-library.pdf/.

[20] ns-3 overview. http://www.nsnam.org/docs/ns-3-overview.pdf/.

[21] Mininet. http://mininet.org/.

[22] Introduction to Mininet. http://cng.ateneo.edu/cng/wyu/classes/cs154/cs154-mn-intro.pdf.

[23] The Click Modular Router Project. http://www.read.cs.ucla.edu/click/click/.

[24] E.Kohler, R.Morris, B.Chen, J.Jannotti, and M. F. Kaashoek. The Click Modular Router. ACM Trans. Comput. Syst. 18, 3 (August 2000), 263-297.

[25] OPNET. http://www.riverbed.com/products-solutions/products/network-planning-simulation/.

[26] OPNET University Program. http://www.opnet.com/university_program/index.html/.

[27] OPNET Overview and Examples. http://utopia.duth.gr/ rdunayts/pdf/guest/OPNET.pdf.

[28] OpenFlow.http://www.openflow.org/.

[29] N.McKeown, T.Anderson, H.Balakrishnan, G.Parulkar, L.Peterson, J.Rexford, S.Shenker, and J.Turner. 2008. OpenFlow: Enabling Innovation In Campus Networks. SIGCOMM Comput. Commun. Rev. 38, 2 (March 2008), 69-74.

[30] Software-Defined Networking:The New Norm for Networks. https://www.opennetworking.org/sdn-resources/sdn-library/whitepapers/.

[31] ISO/IEC 9596-1. 1991. Information Processing Systems-Open Systems Inter- connection-Common Management Information Protocol.

[32] B.Heller, R.Sherwood, N.McKeown, The Controller Placement Problem. Proceedings of the first workshop on Hot topics in software defined networks, August 13-13, 2012, Helsinki, Finland.

[33] The GENI Project. http://www.geni.net/.

[34] R.Watson. Timer-Based Mechanisms in Reliable Transport Protocol Connection Management. Computer Networks, page 5:4756, 1981.

[35] Google Protocol Buffers. https://developers.google.com/protocol-buffers/.

[36] Apache log4j. http://logging.apache.org/log4j/.

[37] RINA Media Streaming Application. https://github.com/yuezhu/rina_video_streaming/.

[38] VLC Media Player. http://www.videolan.org/vlc/index.html/.

[39] LIVE555 Streaming Media. http://www.live555.com/liveMedia/.