Grant Agreement No.: 317814

IRATI

Investigating RINA as an Alternative to TCP/IP

Instrument**:** ***Collaborative Project***
Thematic Priority: ***FP7-ICT-2011-8***

MS16 Phase 3 Use cases description and requirements analysis report

Due date of the Milestone: Month 19
Start date of project: January 1st 2013 Duration: 24 months
version: v.1.0

Project co-funded by the European Commission in the 7th Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	✓
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

FP7 Grant Agreement No.	317814
Project Name	Investigating RINA as an Alternative to TCP/IP
Document Name	MS16 Phase 3 Use cases description and requirements analysis report
Workpackage	WP2
Authors	Matteo Biancani (Interoute) Alessandro Predieri (Interoute) Antonios Kollaras (Interoute) Eduard Grasa (Fundació i2CAT) Miquel Tarzan (Fundació i2CAT) Bernat Gastón (Fundació i2CAT) Francesco Salvestrini (Nextworks) Gino Carrozzo (Nextworks) Vincenzo Maffione (Nextworks)
Editor	Matteo Biancani (Interoute)
Reviewers	John Day (Boston University) Dimitri Staessens (iMinds) Sander Vrijders (iMinds)
Version	V1.0

Abstract

This report describes the use cases that will shape the research, development and experimentation activities that will be carried out by IRATI partners during the third phase of the project. The goal of the third phase use cases is to analyze in more detail a couple of scenarios related to the deployment of RINA in the current Internet and the requirement to support existing applications without initially having to modify them. These scenarios are paramount in demonstrating that there is a feasible migration path for RINA which does not require drastic measures for it to be to be gradually adopted by both users and deployed by network service providers.

TABLE OF CONTENTS

Acronyms 6
1 RINA over IP networks 7
1.1 Introduction and motivation 7
1.2 DIFs over IP networks: a use case analysis 7
1.2.1 Analysis of shim DIF operations 7
1.2.2 More complicated deployment scenarios 12
2 Support of legacy applications 15
2.1 Introduction and motivation 15
2.2 Sockets API: a use case analysis 15
2.3 RINA IPC API: a use case analysis 16
2.4 Mapping the sockets API to the RINA IPC API 17
2.4.1 Name resolution and connecting a socket 17
2.4.2 Binding a socket and accepting incoming connections 18
2.4.3 Reading from the socket and writing to the socket 18
2.4.4 Closing the socket 19
3 References 20
4 Appendix 1: analysis of the sockets API 21
4.1 Server workflow 21
4.1.1 Socket creation 21
4.1.2 Socket binding 21
4.1.3 Preparing the socket for listening for connections 22
4.1.4 Accepting incoming connections 22
4.1.5 Reading data from the socket 23
4.1.6 Writing data to the socket 23
4.1.7 Closing the socket 24
4.2 Client workflow 24
4.2.1 Resolving the server's IP address from the domain name 24
4.2.2 Connecting to the server's address and port 25
5 Appendix 2: analysis of IRATI's RINA IPC API 26
5.1 Server workflow 26
5.1.1 Application registration 26

5.1.2 Accepting incoming flows 27
5.2 Client workflow 29
5.2.1 Flow allocation 29
5.2.2 Writing data to a flow 30
5.2.3 Reading data from a flow 31
5.2.4 Deallocating a flow 31

Acronyms

API	Application Programming Interface
ARP	Address Resolution Protocol
DCCP	Datagram Congestion Control Protocol
DHT	Distributed Hash Table
DNS	Domain Name System
DIF	Distributed IPC Facility
EFCP	Error and Flow Control Protocol
IP	Internet Protocol
IPC	Inter Process Communication
IPCP	IPC Process
LAN	Local Area Network
NAT	Network Address Translation
NMS	Network Management System
OS	Operating System
PDU	Protocol Data Unit
POSIX	Portable Operating Systems Interface
QoS	Quality of Service
RINA	Recursive InterNetwork Architecture
SCTP	Stream Control Transmission Protocol
SDU	Service Data Unit
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VPN	Virtual Private Network

1 RINA over IP networks

1.1 Introduction and motivation

The deployment of RINA over IP networks has a number of research/experimental and commercial applications:

- Shows that there is no need for a painful migration procedure (an "Internet flag-day") to deploy RINA. The TCP and UDP protocols can be wrapped with the IPC API in end-hosts or routers that adopt RINA and the traffic transported just like regular TCP or UDP traffic over the current Internet. Normal DIFs can bridge LANs, the public Internet and private IP layers (as illustrated by Figure 1), allowing applications on top to operate without the need of NATs or other middleboxes.
- It provides an easy way of running experiments with large geographical scopes, with very different connectivity graphs, easy to setup.
- Enables the extension of RINA demos to conference venues, linking the testbeds with the demo venue to better show the capabilities of RINA.
- Could enable RINA products as next-generation Virtual Private Networks over public/private IP networks. These VPNs would be fully programmable (addressing, routing, data transfer, authentication, etc.), allowing a high degree of customization to the VPN operating environment. Examples of application of these VPNs would be to support distributed peer-to-peer applications, or to compete with network virtualization products such as VMware NSX [1], or Midokura Midonet [2].

1.2 DIFs over IP networks: a use case analysis

1.2.1 Analysis of shim DIF operations

We start our analysis by studying the simplest deployment scenario: all the systems with RINA software that are going to be part of the shim DIF are running on a host with public Internet addresses, or are all part of the same private IP network, as illustrated by Figure 1.

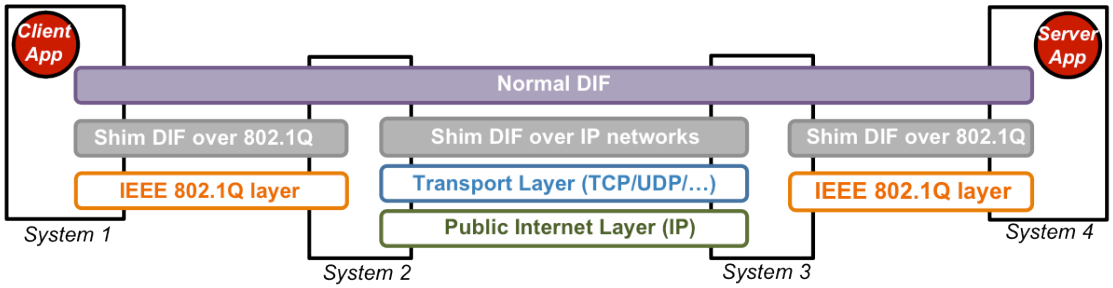

Figure 1 Shim DIF over a single IP Layer

The shim DIF for IP networks has to provide the same service as any other DIF: provide flows so that two or more application instances can communicate over an IP layer – hiding the internal details of the underlying technology (such as IP addresses or transport-layer port numbers). Since it is a shim DIF, the only users should be IPC Processes belonging to other DIFs. Therefore the shim DIF is not required to provide a lot of QoS options such as in-order delivery of reliable transport of SDUs; these functionalities can be provided by higher-level DIFs. ***The goal of the analysis for this shim DIF over IP networks is to identify the minimal shim DIF definition that is complete enough to support the operation of normal IPC Processes***.

Looking at the technological options that the shim DIF for IP networks can use to support the flows it provides, three main alternatives can be explored[1]:

1. *Transport the data from shim DIF flows directly over IP*. This option would avoid the use of the transport layer, putting the shim DIF users' SDUs as payload of the IP datagram. The 'protocol' field in the IP header would indicate that the IP datagram is carrying data from the EFCP protocol, so that the IP stack could differentiate between data belonging to the shim DIF and data belonging to the 'normal' uses of IP. This option is equivalent to the shim DIF over 802.1Q, in that the shim DIF would only be able to provide a single flow between the same pair of IP addresses. Moreover, the use of a non-standard value for the 'protocol' field in the IP header would probably cause many routers, NATs or firewalls to drop the IP packet – making the solution hardly deployable in the real world.

2. *Transport the data from shim DIF flows over UDP*. The second option consists in using UDP to carry the data from the shim DIF flows. As a consequence the shim DIF would be able to support multiple flows (each flow would be mapped to a different UDP port), and it would be easier to identify the shim DIF traffic so that firewalls or NATs could be adequately configured not to filter it (or NAT-traversal techniques used when that would be not possible, see section 2.2.). UDP is a simple protocol that does not provide flow allocation, flow control or congestion control. The lack of flow allocation is an issue that the shim DIF has to overcome somehow, but the lack of flow or congestion control is not a big issue, since the shim DIF is designed to support other DIFs, not applications directly. When using UDP, flow and congestion control can be provided at upper levels, and the shim DIF will not adversely affect higher-level congestion/flow control mechanisms (as it would be the case if TCP was used).

3. *Transport the data from shim DIF flows over TCP*. The third option is about using TCP to carry the data from the shim DIF flows. In contrast with the second solution the shim DIF could exploit some TCP features such as TCP's explicit connection setup (can be mapped to the shim DIF flow allocation), or in-order and reliable delivery of user data. However using

[1] There are other not so popular transport protocols such as SCTP, which could be also considered. SCTP is not an interesting option for the same reasons as TCP (SCTP congestion avoidance procedure is similar to that of TCP).

TCP would have its drawbacks also: TCP's flow and congestion control would limit the throughput achievable by a single flow, and could also interact destructively with the flow and congestion control mechanisms of higher-level DIFs using the flow. Last but not least, NAT transversal techniques are slightly more complicated and less successful when using TCP.

The first approach is clearly not an option since it would be hardly usable in the real world, both for experimentation and production purposes; therefore the discussion is whether the shim DIF over IP networks should use the second approach, the third approach or both (and advertise two QoS cubes with different characteristics for the flows that have to be mapped to UDP and the flows to be mapped to TCP).

In order to take a good decision, we will take into account a couple of considerations. First of all, the goal of the shim DIF is to put a veneer as light as possible over an existing technology to wrap it with the DIF API. A shim DIF is not to be used by applications, only by IPC Processes, therefore its functionality can be limited.

Second, although in controlled environments and for experimental purposes it is feasible to open multiple TCP or UDP ports to support multiple flows, production deployments are another story. It would be much easier to manage the deployment of shim DIFs for IP networks and to lower the reluctance of network administrators for that deployment to happen if the number of UDP or TCP ports required for the shim DIF operation could be minimized. In order to do so the shim DIF could be listening to a single port for incoming requests, which would limit the number of flows that the shim DIF could provide between the same pair of IP addresses to one. This limitation is not much of an issue if UDP is used but it can be with TCP – specially if the shim DIF is over the public Internet - since the throughput of a single TCP flow is limited by the round trip time between the two TCP endpoints (due to the sliding-window flow control techniques used by TCP).

Therefore the most reasonable solution for a shim DIF over IP networks seems to be using **UDP and IP**, and to limit the number of flows between the same pair of IP addresses to one (to minimize the number of ports required for the shim DIF operation). Other features that could be provided by the shim DIF if TCP was used – such as flow control, in-order and reliable delivery of SDUs – can be provided by higher-level DIFs. With this assumption in mind, the following subsections analyze the flow allocation and application registration operations.

1.2.1.1 Application registration

An IPC Process with name "b-1" (application name – application instance) wants to register to the shim DIF for IP Networks. Registration should make the IPC Process "b-1" discoverable by other shim IPC Process in the shim DIF, and allow other IPCPs using the shim DIF to allocate a flow to it. In order to successfully register "b-1" in the shim DIF, "b-1" must be associated to the IP address of the shim IPC Process it will be registered at, as well as to the UDP port where

the shim IPC Process is listening. If the shim DIF allowed for the registration of multiple IPCPs, multiple UDP ports could be used (one for each registered application), but as discussed in section 2.2.1 the shim DIF only allows one IPCP to be registered at each shim IPC Process.

The information about this binding <IPCP name, IP address/UDP port> must be discoverable by the other shim IPC Process in the shim DIF, so that they can properly allocate flows to "b-1" (see section 2.2.1.2). In essence, the shim DIF has to use a distributed directory that maps IPCP application names to IP addresses and UDP ports. There are two main approaches that can be followed to design and implement this directory:

- Use a custom one. For example broadcasting registrations to all the shim DIF members, or using an "ARP-like" approach (broadcasting requests for a mapping) or manually updating the registration information; or use a Distributed Hash Table (DHT) to distributed the directory information. This option should be avoided whenever possible since the goal of the shim DIF is to perform as little as possible and reuse mechanisms from the underlying technology being wrapped by the shim DIF.
- Use the DNS (Domain Name System) infrastructure. In order to do so the IPCP name can be treated as a domain name (e.g. "b-1.i2CAT.net"). A SRV (service locator) record can be inserted in the relevant DNS server in order to map the domain name to an IP address and port. Although this may be a reasonable approach for production deployments, it may not be suitable for experimentation (because access to a DNS server may not be possible). An alternative is to introduce the mappings manually at each system with an IPC Process, so that the shim IPCP code can still use the same logic (DNS resolvers usually look at the mappings that are locally known by the OS before asking a DNS server; these locally-known mappings are caches of the latest resolutions and also learned from configuration files).

Keeping in mind the philosophy of having a minimal functionality in the shim DIF and minimizing deployment overhead, one possible solution is that all the shim IPC Processes in the same shim DIF use the same UDP port number (both for sending and receiving data). This would allow a further simplification of the directory design, since there would be no need to discover the UDP port and would facilitate the use of the DNS infrastructure. Specifically:

- The use of DNS address records (A in case of IPv4 or AAAA in case of IPv6) would be sufficient for successfully carrying out flow allocation.
- If DNS servers were not available, manually adding the domain name to IP address mapping in one of the OSs configuration file would suffice (for example 'etc/hosts' in Linux).

Furthermore, it would be straightforward from an administrative point of view to separate UDP ports belonging to different shim DIFs – since the UDP ports would be part of the shim DIF definition. This would not only facilitate the setup and monitoring of experiments, but also the

administration of shim DIFs in production environments as well as its deployment in scenarios involving one or more NATs.

1.2.1.2 Flow allocation

Two IPC Processes with names "*a-1*" and "*b-1*" (application name – application instance) want to establish a flow through a shim DIF over IP networks, as depicted in Figure 2. The flow provided by the shim DIF will be unreliable, with no guarantees on the order of the data, and with no flow control; but since the users of it are two IPC Processes, the flow characteristics are sufficient.

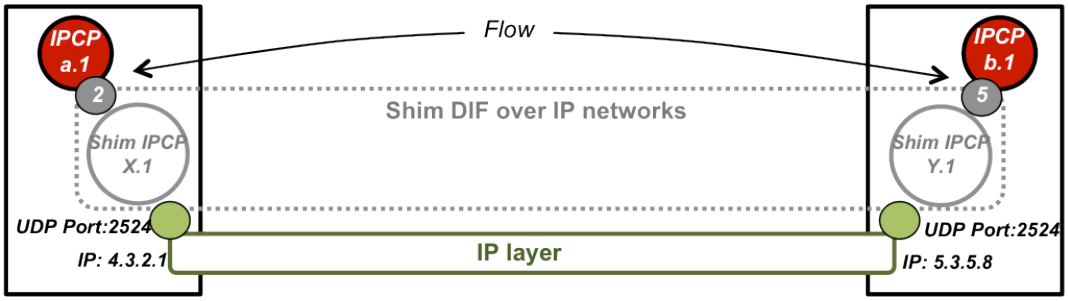

Figure 2 Flow provided by a shim IPC Process over IP Layers

To establish the flow the IPC Process *"a.1"* invokes the shim DIF's API and requests the allocation of a flow to *"b.1"*. The first thing that the shim IPC Process needs to do is to map "b.1" to an IP address (v4 or v6, depending on the shim DIF); the UDP port number is already known since it is part of the DIF definition. As discussed in the previous section, the use of DNS infrastructure is the preferred way of obtaining this mapping (using address records registered at DNS servers or local configuration files in the systems that are part of the shim DIF).

Once the IP address of the shim IPC Process is known, the IPC Process could either consider the flow allocated and respond positively or try to contact the other IPC Process to perform a flow allocation. In the spirit of keeping the shim DIF functionality to the bare minimum, the first option is selected and therefore IPC Process *"b.1"* is not yet aware of the new flow request. When the IPCP *"a.1"* writes the first SDU to the flow, the shim IPCP *"x.1"* assembles a UDP datagram targeted to the *"5.3.5.8"* IP address and port *2524*. When the shim IPC process "y.1" receives it, it can perform a number of checks:

- First it can check if the datagram's source UDP port is the port of the shim DIF, and discard the datagram otherwise.
- If the shim IPC Process knew the IP addresses of all the shim IPCPs that are members of the DIF it could check if the address is one of the expected ones and discard the datagram otherwise. This does not replace the fact that *"a.1"* and *"b.1"* should authenticate if they do not trust each other.

Once the checks are done, the shim IPCP examines if there is a flow to the source IP address of the datagram already allocated. In case it is not, it could try to resolve the name of the IPCP requesting the flow by performing a reverse DNS lookup. With or without the last step, the shim IPCP informs IPCP *"b.1"* about the incoming flow request. If the request is accepted the shim IPCPs just keep generating/parsing UDP PDUs in response to read/write calls to the flow. If the request is denied, the shim IPCP drops the UDP datagram.

A problem that arises is how to deal with the UDP datagrams generated from the "*a.1*" calls to the flow API, since "*a.1*" does not know that "*b.1*" rejected the flow. At some point the IPCP "a.1" will realize that the flow is not working properly (since no data is received) and will deallocate the flow, but meanwhile, what should the shim IPC Process "y.1" do with the datagrams it receives? There seem to be at least three options:

- Treat each UDP datagram as a new flow allocation request, so that IPC Process *"b.1"* would have to reject the incoming flow for every UDP datagram from *"a.1"* received (until *"a.1"* stops sending them). Depending on the frequency at which UDP datagrams are received this could cause a lot of overhead to the shim IPC Process *"y.1"* and the IPC Process *"b.1"*, leading to a Denial of Service attack.
- Blacklist the IP address of "*x.1*" and drop all the UDP datagrams received from it (until the system administrator decides to revert the situation for example). This would not allow IPCP "b.1" to change its mind and accept flow allocation requests from "a.1" (which could be the case if rejecting the first flow allocation request had been an error).
- Blacklist the IP address of "x.1" for a certain amount of time (configurable per shim DIF), and after that time, process UDP datagrams of "x.1" the normal way. Selecting a proper time interval should balance the disadvantages of the former two cases.

1.2.1.3 Flow deallocation

Continuing with the example of Figure 2, at some point IPCP "b.1" or IPCP "a.1" will decide to deallocate the flow. To do so the IPCP will invoke the "deallocate" operation of the shim DIF API, causing the shim DIF to clear all the data structures allocated to the flow. Flow deallocation is a local procedure, meaning that there is no exchange of information between the shim IPC Processes at the endpoints of the flow. Therefore, it could happen that IPCP "a.1" deallocated the flow, but IPCP "b.1" didn't do it. Then, it could happen that "b.1" continued writing data to the flow, resulting in a situation equivalent to the one discussed when the flow was rejected by the application (therefore the same options would be available).

1.2.2 More complicated deployment scenarios

In order to maximize the usefulness and applicability of RINA deployments over IP, there is a need to consider other deployment scenarios that are more complex but rather usual in reality. The deployment scenarios to consider are:

1. Some systems are in a private IP network behind a Network Address Translation (NAT) device, others have public Internet addresses, as illustrated in Figure 3.
2. Systems are in different private IP networks (behind NATs), which are interconnected together via the public Internet, as illustrated in Figure 4.

We will start by assuming that the systems that run the NATs are not under the control of the RINA deployment, and therefore no RINA software can be installed in them. In the first scenario (Figure 3), we can identify two cases.

a) The shim IPCP running in the system with a public IP address (System 3) can just register as explained in section 2.2.1.1. When the shim IPCP running at the system with the private IP address (System 2) tries to establish a flow to the shim IPCP at System 3 it will just work as described in section 2.2.1.2, since the IPCP at System 2 has connectivity to the public Internet (assuming the UDP port used by the shim DIF is not blocked by a firewall).

b) The problematic cases are when the shim IPCP at system 2 needs to register, or when the shim IPCP at System 3 needs to allocate a flow to the shim IPCP at system 2. The NAT translates the private IP address of the interface where the shim IPCP is listening to a public IP address and port. This mapping may be dynamic or change from time to time, therefore the IPCP at system 2 needs a way to discover the public IP address and port used by the NAT.

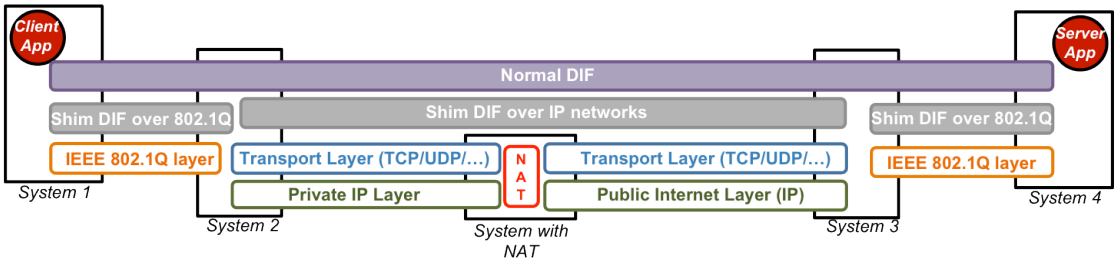

Figure 3 Shim DIF over a private IP layer and the public Internet, interconnected with a NAT

The situation depicted in Figure 4 is still more complex, since now both shim IPCPs are in private IP layers behind two NATs; therefore each shim IPCP needs to discover its public IP address and port.

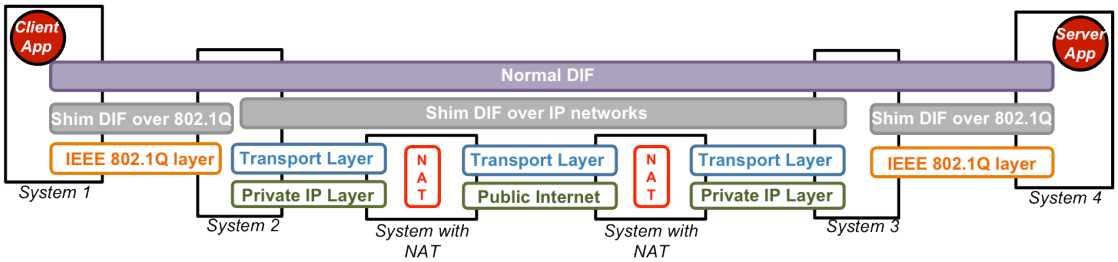

Figure 4 Shim DIF over two private IP layers and the public Internet, interconnected with NATs

There are a number of so-called NAT traversal techniques that can be used by the shim IPCPs to discover if they are behind a NAT and what is their public IP address and port. STUN [3], Session Transversal Utilities for NAT, is a standardised set of methods and a network protocol that enables a host to get this information for some types of NATs. STUN relies on a server reachable through the public Internet (as illustrated in Figure 5). The basic mechanism is the following: a STUN client sends a query to the server, which replies with the IP address and UDP port that it has seen in the packet received from the client. Once the shim IPCP has learned the public IP address and port used by the NAT (using STUN or other techniques), it can proceed with the normal operation of the shim DIF as described in section 2.2.1.

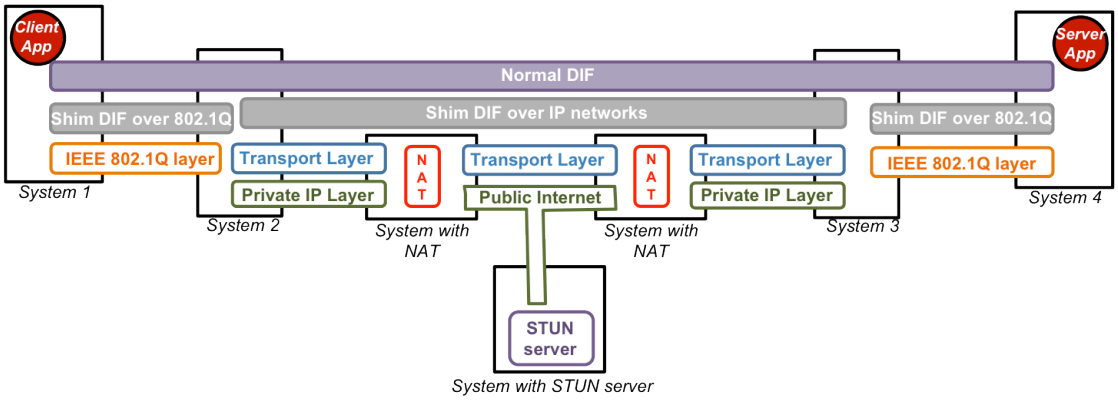

Figure 5 Allowing System 2 and System 3 to communicate via UDP hole punching (using a STUN server)

If RINA software can be installed on the NAT systems (as shown in Figure 6), each IP layer can be encapsulated by a shim DIF, avoiding the need for using NAT transversal techniques. Each shim DIF would have the scope of a single IP layer, and a normal DIF on top of the different shim DIFs would provide IPC services over them.

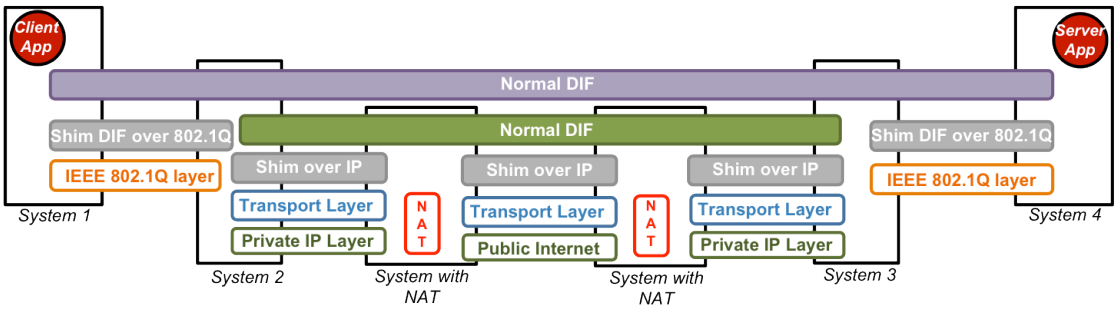

Figure 6 DIF deployment over IP layers installing RINA software in systems with NATs

2 Support of legacy applications

2.1 Introduction and motivation

In order to fully exploit the benefits of the RINA architecture, applications have to be written to use the native RINA IPC API. This exercise requires a modification of the application's source code, which can be costly depending on the usage of the sockets API by the application.

Having to adapt existing applications to the RINA IPC API or to develop native RINA IPC API applications quickly becomes a very time-consuming activity. Therefore a mechanism that allowed existing applications to run over RINA DIFs without having to modify them is a very interesting goal both for RINA demonstration and experimentation activities. This is the goal of the **faux-sockets API**: intercept application calls to the sockets API and redirect them to the RINA IPC API.

2.2 Sockets API: a use case analysis

Current applications use the sockets API in order to communicate with other applications locally (the case of Unix sockets) or via an IP network using the transport services provided by UDP and TCP (although these are the most use transport protocols, SCTP – Stream Control Transmission Protocol - and DCCP – Datagram Congestion Control Protocol - are also available via the Sockets API). This report will focus on the use case of applications communicating via an IP network.

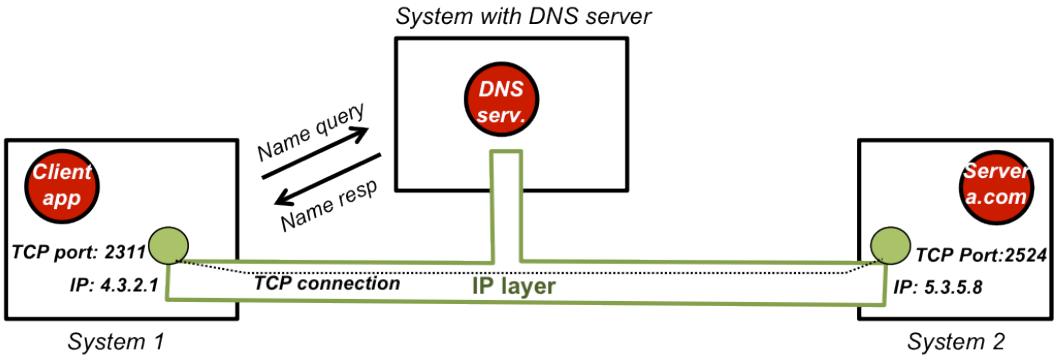

Figure 7 A client application communicating to a server application via the sockets API

Figure 7 illustrates a simple scenario in which a client application establishes a TCP connection to the "a.com" server application. Since the sockets API only accepts IP addresses to define the endpoints of a socket, the client application first needs to resolve the name of the server "a.com" into one of the IP addresses that the server may be listening to. This resolution is usually performed by resolver libraries that interact with the Domain Name System (DNS) infrastructure or use local Operating System knowledge to return an address.

The analysis of the sockets API can be partitioned into the functionality invoked by the server and the functionality invoked by the client. Appendix 1 analyzes the usual workflows for server and client applications; highlighting the calls to the sockets API as defined by the POSIX (Portable Operating System Interface) standards [4].

2.3 RINA IPC API: a use case analysis

Applications supporting RINA use the IPC service API in order to communicate with other applications by name regardless of their location. Figure 8 illustrates a simple scenario in which a client application ("a.1") establishes a flow to the "b.com" server application. The DIF internally resolves the address of the IPC Process through which the "b.1" application is reachable; therefore there is no need for an external name to address resolution –as it is the case with the sockets API.

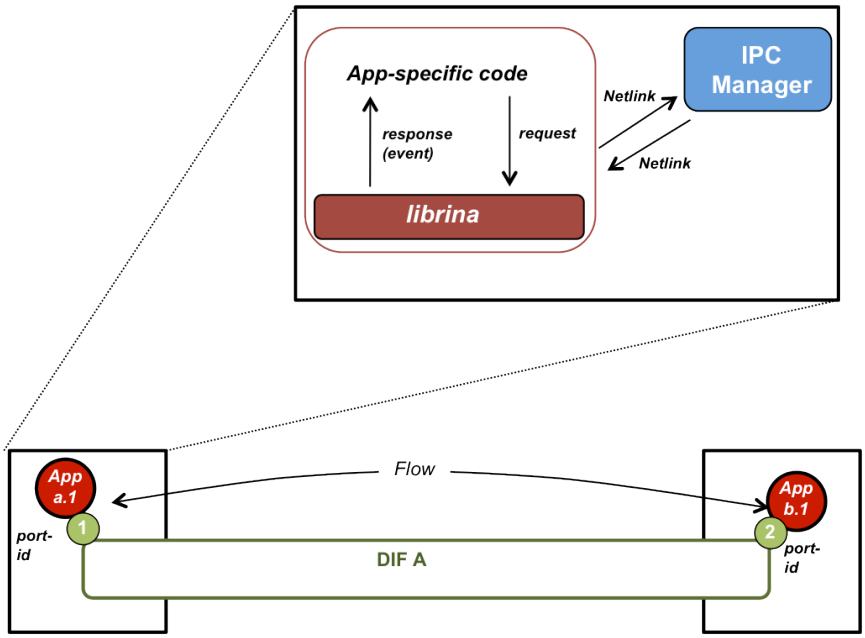

Figure 8 A client application communicating to a server application via the IPC service API

In order to make the comparison with sockets easier, we will divide the analysis of the IPC service API into the functionality invoked by the server and the functionality invoked by the client.

The current IPC API implemented by IRATI is event-based and non-blocking: the application sends a request to the IPC Manager [5], which then causes the appropriate action on one or more DIF(s) to happen. For each operation requested to the IPC Manager, the application receives a unique identifier that will be later used to identify the IPC Manager response. Appendix 2 provides the details of the analysis of IRATI's RINA IPC API.

2.4 Mapping the sockets API to the RINA IPC API

A full analysis of this issue will be part of the specification of the faux sockets API, but this report discusses initial ideas and considerations.

2.4.1 Name resolution and connecting a socket

The sockets API works with socket addresses. In the case of the Internet, those are IP addresses that belong to different interfaces. Applications can choose to directly work with IP addresses (which is a bad practice, since the application is then coupled with one of the interfaces of the system where it is executing), or work with domain names that are resolved to IP addresses through the use of the Domain Naming System, DNS. Only the latter applications are initially considered for faux-sockets support.

The RINA API works with application names. Application names are internally resolved by the DIF to an IPC Process address; therefore IPC Process addresses are not externally exposed and applications don't need to perform an external name resolution – as in the case of the sockets API. Client applications written to the sockets API usually perform these steps:

1. Resolution of the target application domain name to an "Internet address" via *getaddrinfo* function.
2. Connecting the socket to the remote application by passing the *sockaddr* struct returned by the call to the *getaddrinfo* function.

The faux-sockets API has to implement the *getaddrinfo* and use it to transform the information passed by the application (domain name and port) into a RINA application name. This name can be returned as a struct that can be casted to *sockaddr* (for example *sockaddr_rina*), which contains the following attributes:

- **process_name** The name of the application process. The domain name can be used to populate this field.
- **process_instance** The instance of the application process. A zero-length string by default.
- **entity_name** The name of the application entity. The port or service_name instance can be used to populate this name.
- **entity_instance** The instance of the application entity. A zero-length string by default.

Calls to connect can directly be redirected by the faux-sockets API to flow allocation requests, since: i) the destination application naming information will be passed as the *sockaddr* structure (which can be casted to a *sockaddr_rina* structure); ii) a limited set of flow characteristics can be inferred from the "type" argument used to construct the socket (*SOCK_STREAM* for reliable stream-oriented, *SOCK_DGRAM* for datagrams, *SOCK_SEQPACKET* for reliable sequenced packets).

2.4.2 Binding a socket and accepting incoming connections

When an application using the sockets API binds a socket to a certain interface or port, it provides two pieces of information: the address of the interface the socket has to be bound to, as well as the port number. Socket binding operations can be naturally mapped to the application registration procedures in the RINA IPC API (although in RINA applications cannot register by themselves since they do not have enough privileges to do so; some sort of management system like the OS or an NMS must do it on their behalf). The two questions that need to be answered to produce a complete mapping are:

- What is the naming information of the application being registered?
- In what DIFs does the application need to be registered?

An application written to the sockets API specifies the port the socket has to be bound to. This information can be used to partially fill the application naming information, by assigning it to the application entity name as discussed in section 2.4.1. However, the application name is still missing. Both the application process name and the application process instance don't change during the application process lifetime (since it is instantiated until it is destroyed), therefore both pieces of information can be part of the static information used to configure the faux sockets API. A way to populate this information that is consistent with the "name resolution" and "connecting to a socket" procedure described in section 2.4.1 is to use a domain name to populate the application process name field.

In order to answer the second question, let's take a look at the behaviour of the sockets API. The application can request the socket to be bound to a specific interface – and therefore just accept incoming connections from that interface –, or to accept connections from all the interfaces. In the former case the application will pass the IP address of the interface, in the latter it will pass the *INADDR_ANY* constant. This behaviour is somewhat equivalent to applications registering on a single, specific DIF – whose name is provided as part of the registration request – or to applications registering in all the DIFs that are visible to them. However IP addresses are not DIF names, therefore if this behaviour is to be supported there must be a local mapping from the IP address passed by the application to a DIF name.

Once applications are registered, incoming flow requests cannot be accepted until the application has invoked the "listen" function on the socket; until this happens all incoming flow requests have to be rejected. After "listen" is invoked, incoming flow requests can be queued by the faux-sockets library until the application starts invoking the "accept" function on the socket. As a result a new socket will be created, which will be internally mapped to a "Flow" object by the faux-sockets library.

2.4.3 Reading from the socket and writing to the socket

The mapping of the read and write operations from sockets to the IPC API is straightforward since they use the same paradigm: passing buffers that will be populated with the data read

from the socket/flow (in the case of read operations), or passing buffers that contain the data to be written to the socket/flow (in the case of write operations).

2.4.4 Closing the socket

The “close” and “shutdown” operations can be used to fully or partially release the resources assigned to a socket. The faux-sockets library should map both operations to calls to the “deallocate_flow” function, which is the equivalent operation in the RINA IPC API. However the current implementation of the “deallocate” operation in IRATI’s prototype is not granular enough to allow for the partial deallocation of just the “read” or just the “write” operations of the flow. Therefore calls to “shutdown” can only be supported if both the “read” and “write” components of the socket are to be released at the same time.

3 References

[1] VMware NSX, available online at http://www.vmware.com/products/nsx

[2] Midonet network virtualization overview, available online at http://www.midokura.com/midonet/

[3] J. Rosenberg, R. Mahy, P. Matthews, D. Wing. "Session Traversal Utilities for NAT (STUN)", RFC 5389. October 2008.

[4] IEEE Std 1003.1-2008 and The Open Group Technical Standard Base Specifications, Issue 7. "POSIX-1.2008 specification", available online at http://pubs.opengroup.org/onlinepubs/9699919799/

[5] S. Vrijders, D. Staesses, D. Colle, F. Salvestrini, E. Grasa, M. Tarzan and L. Bergesio "Prototyping the Recursive Internetwork Architecture: The IRATI Project Approach", IEEE Network, Vol. 28, no. 2, March 2014

4 Appendix 1: analysis of the sockets API

4.1 Server workflow

A typical scenario for a TCP server using the sockets API involves creating a socket, binding it to a local address/port, prepare the socket to start listening for incoming connections, accept the incoming connections, perform some work reading/writing data to the accepted connection and closing the socket once the task is over.

4.1.1 Socket creation

Creates an endpoint for a communication and returns the file descriptor for that endpoint. Function prototype:

```
int socket(int domain, int type, int protocol)
```

- **domain** specifies the protocol family of the created socket: *AF_INET* for IPv4, *AF_INET6* for IPv6 or *AF_UNIX* for UNIX sockets.
- **type** specifies the type of service the socket provides: *SOCK_STREAM* for reliable stream-oriented, *SOCK_DGRAM* for datagrams, *SOCK_SEQPACKET* for reliable sequenced packets and *SOCK_RAW* for raw access to the IP layer.
- **Protocol** specifies the actual transport protocol to use: *IPPROTO_TCP* for TCP, *IPPROTO_UDP* for UDP, *IPPROTO_SCTP* for SCTP, etc.

Returns a positive integer representing the socket descriptor or -1 on error (setting *errno* to indicate the error). Example (TCP socket for IPv4):

```
int socket_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP).
```

4.1.2 Socket binding

Assigns a socket to an address. Function prototype:

```
int bind(int sockfd, const struct sockaddr * addr, socklen_t addrlen)
```

- **sockfd** the descriptor of the socket to perform the bind on.
- **sockaddr** a pointer to a socket address structure, representing the address to bind to.
- **addrlen** size of the sockaddr “derived” structure.

The purpose of struct *sockaddr* is to act as an abstract base class, and so to cast pointers to derived classes to pointers to the base class. For example, the IPv4 socket address structure – called *sockaddr_in* – contains the following fields:

- **short sin_family** the family of the address (*AF_INET*).
- **unsigned short sin_port** the port number where the socket will be bound to.
- **struct in_addr sin_addr** the IP address the socket will be bound to (*INADDR_ANY* will bind the socket to all the IP interfaces available at the machine).

'Bind' returns 0 on success or -1 if an error occurs (setting *errno* to indicate the error). Example:

```
struct sockaddr_in sock_addr;
…
sock_addr.sin_family = AF_INET;
sock_addr.sin_port = htons(1100);
sock_addr.sin_addr.s_addr = htonl(INADDR_ANY);
bind(socket_fd,(struct sockaddr *)&sock_addr, sizeof(sock_addr));
```

4.1.3 Preparing the socket for listening for connections

Prepares a socket for accepting incoming connections (only necessary for stream-oriented data modes). Function prototype:

```
int listen(int sockfd, int backlog)
```

- **sockfd** the descriptor of the socket that has to listen for connections.
- **backlog** an integer representing the number of pending connections that can be queued at anytime. Once a connection is accepted, it is dequeued.

Returns 0 on success, -1 on error (setting *errno* to indicate the error). Example:

```
listen(socket_fd, 10);
```

4.1.4 Accepting incoming connections

Creates a new socket for an incoming connection and removes the connection from the listen queue. Function prototype:

```
int accept(int sockfd, struct sockaddr *cli_addr, socklen_t *addr_len)
```

- **sockfd** the descriptor of the socket that has to accept incoming connections
- **cli_addr** pointer to the structure where the other endpoint's address will be copied
- **addr_len** pointer where the length of the other endpoint's address structure is returned

Returns the new socket descriptor for the accepted connection or -1 on error (setting *errno* to indicate the error). Example:

```
struct sockaddr_in cli_addr;
sock_len_t cli_len;
…
clilen = sizeof(clid_addr);
int client_sd = accept(socket_fd, (struct sockadd *)&cli_addr,
&cli_len);
```

4.1.5 Reading data from the socket

The sockets API defines three functions to read data from a socket: *rcv*, *rcvfrom* and *rcvmsg*. The *rcv* call is normally only used in connection (stream) – oriented sockets, and is equivalent to calling *rcvfrom* with a null source address argument. Function prototypes:

```
ssize_t rcv(int sockfd, void * buf, size_t len, int flags)

ssize_t rcvfrom(int sockfd, void * buf, size_t len, int flags, struct
sockaddr * src_addr, socklen_t * addr_len)

ssize_t rcvmsg(int sockfd, struct msghdr * msg, int flags)
```

- **sockfd** the descriptor of the socket to be read
- **buf** a pointer to store the data available in the socket
- **len** the maximum number of bytes to read
- **flags** options that may affect the execution of the read operation

All three operations return the number of bytes read if successful, or -1 if an error occurred (setting *errno* to indicate the error). Example:

```
#define RCVBUFSIZE 32
char buffer[RCVBUFSIZE];
int bytes_rcvd = rcvfrom(client_sd, buffer, RCVBUFSIZE, 0);
```

4.1.6 Writing data to the socket

The sockets API defines three functions to write data to a socket: *send*, *sendto* and *sendmsg*. The *send* call is normally only used in connection (stream) – oriented sockets, and is equivalent to calling *sendto* with a null destination address argument. Function prototypes:

```
ssize_t send(int sockfd, void * buf, size_t len, int flags)

ssize_t sendto(int sockfd, void * buf, size_t len, int flags, struct
sockaddr * dest_addr, socklen_t * addr_len)

ssize_t sendmsg(int sockfd, struct msghdr * msg, int flags)
```

- **sockfd** the descriptor of the socket to be read
- **buf** a pointer to the data to be written to the socket
- **len** the number of bytes to write
- **flags** options that may affect the execution of the read operation

All three operations return the number of bytes sent if successful, or -1 if an error occurred (setting *errno* to indicate the error). Example:

```
#define SENDBUFSIZE 32
char send_buffer[SENDBUFSIZE];
int bytes_sent = send(client_sd, send_buffer, SENDBUFSIZE, 0);
```

4.1.7 Closing the socket

The *close* function releases all the operating system resources assigned to the socket. On the other hand, the *shutdown* function enables for a finer control over the socket functionality, enabling the caller to decide if it wants the socket to stop sending data, stop receiving data or both. Function prototype:

```
int close(int sockfd)

int shutdown(int sockfd, int how)
```

- **sockfd** the descriptor of the socket that has to be closed or shut down.
- **how** specifies the type of shutdown: *SHUT_RD* disables further receive operations, *SHUT_WR* disables further write operations, and *SHUT_RDWR* disables both.

Both functions return 0 on success and -1 on error (setting *errno* to indicate the error). Example:

```
shutdown(client_sd, SHUT_WR);
close(client_sd);
```

4.2 Client workflow

A straw-man sockets client that requests a reliable stream service performs the following steps: resolving the server's IP address from the domain name, connecting to the server, reading and writing data and finally closing the socket.

4.2.1 Resolving the server's IP address from the domain name

The *getaddrinfo* function returns one or more *addrinfo* structures that are associated to a domain name and/or a port number (combining the functionalities provided by the old gethostbyname and getservbyname functions). Function prototype:

```
int getaddrinfo(const char* domain_name, const char* service, const
struct addrinfo *hints, struct addrinfo ** res)
```

- **domain_name** is the domain name of the host that hosts the service.
- **service** is the transport port number or a string that can be mapped to a port number.
- **hints** is a pointer to an *addrinfo* structure that may be passed as a filter in order to limit the results returned by the function (only return addresses belonging to a certain family, or protocol, for example).
- **res** a pointer to a linked list with the resolved addresses. More than one result os possible since the network host may be multihomed, or the same service is available from multiple socket types, for instance.

The addrinfo structure has the following fields:

- **int ai_family** the address family: *AF_INET*, *AF_INET6* or *AF_UNIX.*
- **int ai_socktype** the socket type: *SOCK_STREAM*, *SOCK_DGRAM*, etc.

- **int ai_protocol** the transport protocol underlying the socket: *IPPROTO_TCP*, *IPPROTO_UDP*, etc.
- **int ai_flags** extra options.
- **socklen_t ai_addrlen** the length of the sockaddr field**.**
- **struct sockaddr * ai_addr** the actual address information.
- **char * ai_canonname** canonical name of the node.
- **struct addrinfo * ai_next** a pointer to the next resolved address (if any).

The function returns zero on success, or an error code if it fails.

4.2.2 Connecting to the server's address and port

The *connect* function attempts to make a connection on a connection-mode socket or to set or reset the peer address of a connectionless-mode socket. Function prototype:

```
int connect(int sockfd, const struct sockaddr *address, socklen_t len)
```

- **sockfd** is the descriptor of the socket to be connected.
- **address** points to an address structure containing the peer address.
- **len** is the length of the structure containing the peer address.

The *connect* function returns 0 on success and -1 on error (setting *errno* to indicate the error).

5 Appendix 2: analysis of IRATI's RINA IPC API

5.1 Server workflow

A typical scenario for a server application that uses the IPC service API is to get the application registered in one or more DIFs (which should be performed by an entity with enough privileges, such as the Operating System or another type of Management System), to accept/reject incoming flow requests from different applications, perform read/write operations on the accepted flows and deallocate them once they are no longer useful.

5.1.1 Application registration

Requests the registration of an application instance to one ore more DIFs to the IPC Manager.

```
unsigned int IPCManager::requestApplicationRegistration(const
ApplicationRegistrationInformation& app_reg_info)
```

- **app_reg_info** a constant reference to the class that contains the application registration information.

The application registration information class has the following members:

- **appName** The naming information of the application (entity) to be registered, including: process name (compulsory), process instance, entity name, entity instance.
- **type** specifies the type of registration: *SINGLE_DIF* to register the application to a specific DIF, *ANY_DIF* to register the application to a DIF chosen by the IPC Manager.
- **difName** If the type is *SINGLE_DIF*, this variable contains the name of the DIF the application should be registered at.

Returns a positive unsigned integer representing the correlation id of the response message. When the IPC Manager has sent the answer to the requestor, the application will receive an event informing it about the result. The event will be encapsulated in an object of type *RegisterApplicationResponseEvent*, with the following information:

- **sequenceNumber** The id of the response message, so that it can be correlated with the request message.
- **result** An integer representing the result of the operation: 0 indicates success, a negative integer provides an error code.
- **apName** The naming information of the registered application.
- **difName** The name of the DIF where the application was registered.

Once the client has received the response, it has to call the following operations to update the internal state of librina (since it keeps internal state on behalf of the operation like the list of application registrations, list of allocated flows, etc.). If the operation has been unsuccessful:

```
void IPCManager::withdrawPendingRegistration(unsigned int seqNumber)
```

- **seqNumber** the id of the pending registration.

If the operation has been successful:

```
ApplicationRegistration     *     IPCManager::commitPendingRegistration(
unsigned int seqNumber,  const ApplicationProcessNamingInfo& difName)
```

- **seqNumber** the id of the pending registration.
- **difName** the name of the DIF where the application has been registered.

The operation returns a pointer to an ApplicationRegistration class, which contains all the DIFs where a particular application entity has been registered. This information can be accessed by the application at any moment by invoking the following operation:

```
std::vector<ApplicationRegistration                                    *>
IPCManager::getRegisteredApplications()
```

Example:

```
IPCManager * ipcManager;
ApplicationRegistrationInformation app_reg_info;
unsigned int id;

… (obtain reference to IPC Manager) …

app_reg_info.appName.processName = “b”;
app_reg_info.appName.processInstance = “1”;
app_reg_info.type = SINGLE_DIF;
id = ipcManager->requestApplicationRegistration(app_reg_info);


… (other stuff done meanwhile, response event arrives) …

RegisterApplicationResponseEvent event;
if (event.result == 0)
    ipcManager->commitPendingResitration(id, event.difName);
else
    ipcManager->withdrawPendingRegistration(id);
```

5.1.2 Accepting incoming flows

When a registered application is the target of a flow request, it will receive an event of type *FLOW_ALLOCATION_REQUESTED*, with the event information encapsulated in an object of type *FlowRequestEvent*:

- **sequenceNumber** The id of the request message, so that it can be correlated with the response message.
- **portId** The port-id assigned to the flow if the flow is accepted.
- **localApplicationName** The name of the application that is the target of the flow.
- **remoteApplicationName** The name of the application that requested the flow.
- **difName** The name of the DIF that will support the flow.
- **flowSpecification** The characteristics of the flow (in-order-delivery, reliability, delay, etc.)

When the application receives this event it can examine the flow allocation request information and decide whether it accepts the flow or not. The application needs to communicate its decision to the IPC Manager via the following operation:

```
Flow * IPCManager::allocateFlowResponse(const FlowRequestEvent & flow_req_event, int result, bool notify_source)
```

- **flow_req_event** is the object containing the information of the flow allocation request.
- **result** 0 if the flow is accepted, another integer indicating an error code otherwise.
- **notify_source** only used if the flow is rejected; true if the source application should be notified about the rejected flow allocation request, false otherwise.

This operation returns a pointer to a *Flow* object, which allows the application to write and read data from the flow (as later explained in the '*Client Workflow*' section).

Example:

```
IPCEvent* event = ipcEventProducer->eventWait();
switch (event->eventType) {
…

case FLOW_ALLOCATION_REQUESTED_EVENT:
   Flow * flow;
   FlowRequestEvent* flow_req_event =
dynamic_cast<FlowRequestEvent*>(event);
   // check fields of flow_req_event and decide if flow if accepted
   …
   // flow is accepted
   flow = ipcManager->allocateFlowResponse(*flow_req_event, 0, true);
  // do something with the flow
   …
  break;

…

}
```

5.2 Client workflow

A typical scenario for a simple client application that uses the IPC service API is to request a flow to another application, perform read/write operations on the accepted flow if accepted and deallocate the flow once it is no longer useful.

5.2.1 Flow allocation

Requests the allocation of a flow to another application by name. The requesting application can specify a number of characteristics for the flow.

```
unsigned int IPCManager::requestFlowAllocation(const
ApplicationRegistrationInformation& local_app_name, const
ApplicationRegistrationInformation& remote_app_name, const
FlowSpecification& flow_spec)
```

- **local_app_name** is the naming information of the application requesting the flow.
- **remote_app_name** is the naming information of the application that is the target of the flow.
- **flow_spec** is the object that captures the characteristics requested for the flow (such as maximum delay, in-order-delivery of data, reliability, average throughput, etc.)

The operation returns a positive unsigned integer representing the correlation id of the response message. When the IPC Manager has sent the answer to the requestor, the application will receive an event informing it about the result. The event will be encapsulated in an object of type *AllocateFlowRequestResultEvent*, with the following information:

- **sequenceNumber** The id of the response message, so that it can be correlated with the request message.
- **portId** If positive, the flow has been accepted by the remote application and this is the port-id assigned of the flow. If negative indicates that the flow has been rejected by the remote application.
- **difName** The name of the DIF that is providing the flow.

Once the client has received the response, it has to call the following operations to update the internal state of librina (since it keeps internal state on behalf of the operation like the list of application registrations, list of allocated flows, etc.). If the operation has been unsuccessful:

```
void IPCManager::withdrawPendingFlow(unsigned int seqNumber)
```

- **seqNumber** the id of the pending flow allocation request.

If the operation has been successful:

```
Flow * IPCManager::commitPendingFlow( unsigned int seqNumber, int
portId, const ApplicationProcessNamingInfo& difName)
```

- **seqNumber** the id of the pending flow allocation request.

- **portId** the local identifier of the flow.
- **difName** the name of the DIF that is providing the flow.

The operation returns a pointer to a *Flow* object, which allows the application to write and read data from the flow. The application can get the pointers to all the currently allocated Flow objects by invoking the following operations:

```
std::vector<Flow *> IPCManager::getAllocatedFlows()
```

Example:

```
IPCManager * ipcManager;
FlowSpecification flow_spec;
unsigned int id;

… (obtain reference to IPC Manager) …

//populate flow_spec object with desired flow characteristics
flow_spec.in_order_delivery = true;
flow_spec.delay = 200;
id = ipcManager-
>requestFlowAllocation(ApplicationProcessNamingInformagion("a", "1"),
ApplicationProcessNamingInformagion("b", "1"), flow_spec);
… (other stuff done meanwhile, response event arrives) …

AllocateFlowRequestResultEvent event;
Flow * flow;
if (event.result == 0)
    flow = ipcManager->commitPendingFlow(id, event.portId,
event.difName);
else
  ipcManager->withdrawPendingFlow(id);
```

5.2.2 Writing data to a flow

In order to write data to a flow, the following function must be invoked.

```
void Flow::writeSDU(void * sdu, int size)
```

- **sdu** is a pointer to a buffer that contains the data to be written.
- **size** is the size of the data to be written, in bytes.

Example:

```
Flow * flow;
```

```
char * buffer = new char[30];
//populate buffer
…
flow->writeSDU(buffer, 30);
```

5.2.3 Reading data from a flow

In order to read data from a flow, the following function must be invoked.

```
int Flow::readSDU(void * sdu, int maxBytes)
```

- **sdu** is a pointer to a buffer that will contain the data read.
- **maxBytes** the maximum number of bytes that can be read from the flow.

The function returns the number of bytes read. Example:

```
Flow * flow;
char * buffer = new char[1000];
int bytesRead = flow->readSDU(buffer, 1000);
```

5.2.4 Deallocating a flow

Once the application is done with the flow and wants to deallocate it, it has to invoke the following function.

```
unsigned int IPCManager::requestFlowDeallocation(int portId)
```

- **portId** is the local identifier of the flow to be deallocated.

The operation returns a positive unsigned integer representing the correlation id of the response message. When the IPC Manager has sent the answer to the requestor, the application will receive an event informing it about the result. The event will be encapsulated in an object of type *DeallocateFlowResponseEvent*, with the following information:

- **sequenceNumber** The id of the response message, so that it can be correlated with the request message.
- **portId** The port-id of the flow to be deallocated.
- **result** 0 if the operation was successful, another integer indicating an error code otherwise.

Once the client has received the response, it has to call the following operations to update the internal state of librina (since it keeps internal state on behalf of the operation like the list of application registrations, list of allocated flows, etc.).

```
void IPCManager::flowDeallocationResult(int portId, bool success)
```

- **portId** the local identifier of the flow.

- **success** true if the flow was deallocated, false otherwise.

Example:

```
IPCManager * ipcManager;
Int port_id

… (obtain reference to IPC Manager) …
id = ipcManager->requestFlowDeallocation(port_id);
… (other stuff done meanwhile, response event arrives) …

DeallocateFlowResponse event;
if (event.result == 0)
 ipcManager->flowDeallocationResult(port_id, true);
else
 ipcManager->flowDeallocationResult(port_id, false);
```