Experimental evaluation of a Recursive InterNetwork Architecture prototype

Sander Vrijders[1], Dimitri Staessens[1], Didier Colle[1]
Francesco Salvestrini[2], Vincenzo Maffione[2]
Leonardo Bergesio[3], Miquel Tarzan-Lorente[3], Bernat Gaston[3], Eduard Grasa[3]
[1]Ghent University - iMinds, INTEC, Gaston Crommenlaan 8 bus 201, 9050 Gent, Belgium
E-mail: {firstname.lastname}@intec.ugent.be
[2]R&D, neXtworks s.r.l., via Livornese 1027, Pisa, Italy
E-mail: {f.salvestrini, v.maffione}@nextworks.it
[3]i2CAT Foundation, Jordi Girona, Barcelona, Spain
E-mail: {firstname.lastname}@i2cat.net

***Abstract*—The Recursive InterNetwork Architecture (RINA) is a recently proposed network architecture based on first principles, which promises to solve a number of issues present in the current Internet such as the lack of inherent security. In this paper, we present the experimental evaluation of the first performance-oriented implementation of RINA, the IRATI stack. Our open source stack is designed for GNU/Linux Operating Systems, with key components developed in kernel space for optimal performance. After briefly introducing the architecture, we present the main features of the stack, give some details about the implementation and discuss some trade-offs that had to be taken into account. We present use case scenarios for the evaluation, which were implemented in a test environment, and present the performance, achieving a goodput close to line rate on a GbE link, even when multiple Distributed Inter Process Communication Facilities (DIFs) are stacked.**

I. Introduction and motivation

The vast amount of effort spent on Future Internet research is a clear indication that there are significant drawbacks present in the current TCP/IP Internet architecture. Mobility of hosts, for instance, is not easily achievable with a normal TCP/IP stack, although several solutions have been proposed to overcome this problem such as Mobile IP [1], Locator/ID Separation Protocol (LISP) [2] and Software Defined Networks (SDN) [3]. The complication in implementing mobility in TCP/IP stems from the location-dependency of the IP addresses. While LISP and Mobile IP try to overcome this, their solution of overloading the semantics of the IP address is very complex and introduces scalability problems [4].

Another important issue in the current Internet is the congestion control/avoidance scheme, which is running end-to-end. Embedding congestion control in TCP maximizes the length of the control loop and its variance, making it the least optimal solution from a control theory perspective [5].

The Internet also has well known issues related to security, Quality of Service (QoS) and multicast [6]; but perhaps the worst problem is the explosion in complexity. The Internet protocol suite keeps steadily growing, as shown by the increase in the number of RFC documents published by the IETF [7], with lots of protocols that are very similar in behaviour but address specific problems in particular environments. This growing complexity makes the system as a whole more expensive to implement and operate, more difficult to upgrade and less predictable, since new protocols can interact with existing ones in unforeseen ways.

These issues combined cause the Internet to run at sub-optimal levels of resources utilization. All this illustrates the need for a well-defined network architecture that encourages networking engineers to apply a disciplined approach towards solving networking problems. Providing such a structured network architecture is a key objective of RINA.

II. RINA in a nutshell

A. Architecture

The Recursive InterNetwork Architecture (RINA) [6] [8] builds on the premise that "networking is Inter-Process Communication (IPC) and IPC only" [9]. Figure 1 shows the RINA architectural model [10]. In contrast to the 7-layer OSI model, where each layer provides a different function, the RINA architecture is based on a single type of layer (Distributed Inter Process Communication Facility, DIF) which can be repeated as many times as required by the network designer. Each DIF is a homogenous distributed application, comprising only Inter Process Communication (IPC) processes, which provide Inter-Process communication services over a given scope to the distributed applications above (which may themselves be DIFs). In RINA, invariant parts (mechanisms) and variable parts (policies) are separated in different components of the architecture. This makes it possible to customize the behavior of a DIF to optimally operate on a certain environment by redefining specific sets of policies, without the need to re-implement the same mechanisms over and over again.

A normal IPC process consists of components that are dedicated to data transfer, data transfer control, and layer management. A shim IPC process constitutes a minimal veneer on top of a legacy communication mechanism (e.g. Ethernet, TCP/IP). All IPC processes, regardless of whether they are normal or shim IPC processes, provide the same interface to their client applications.

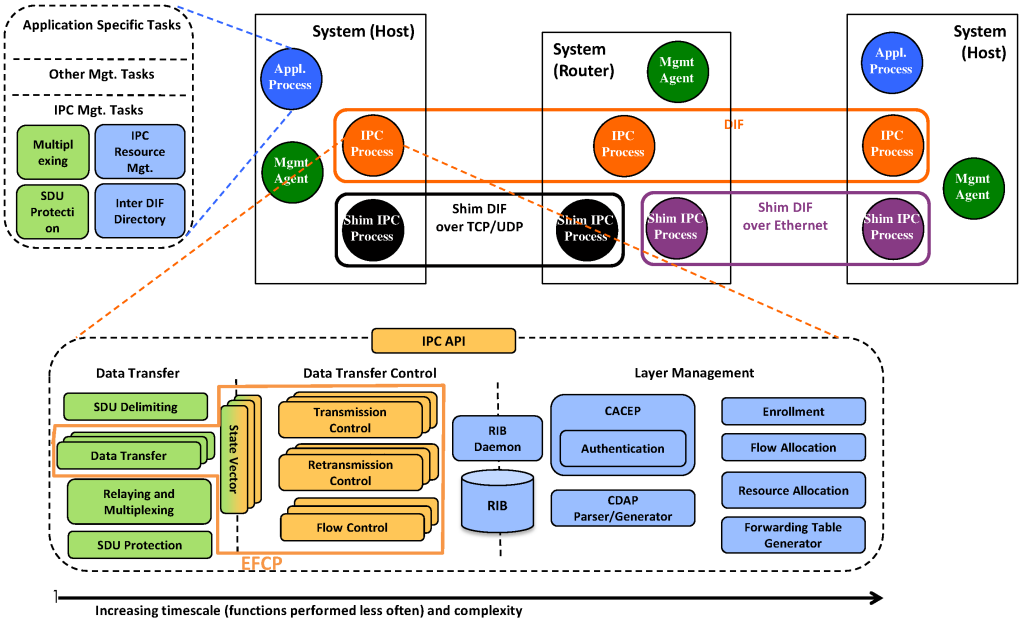


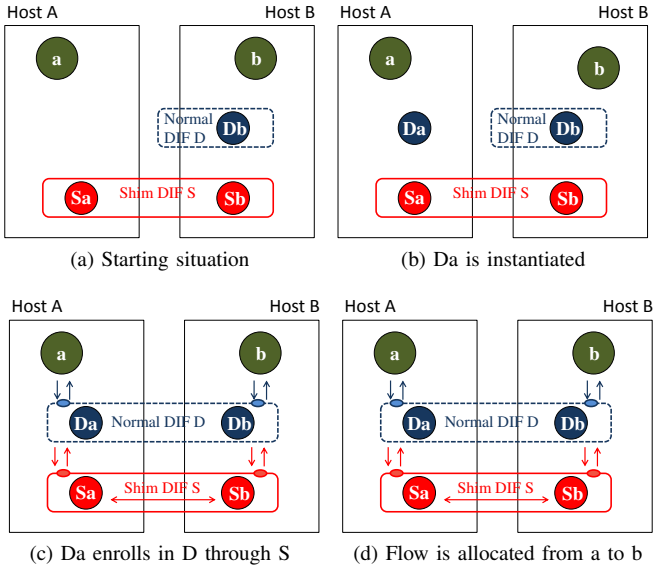
Fig. 1. The RINA Architecture Reference Model. [10]

When an application (or an IPC process) wants to use the services provided by an IPC process, it uses the following API operations:

- *portId allocateFlow(destAppName, List<qosParams>)*. This operation enables an application to allocate a flow to a destination application (identified by destAppName), specifying a list of desired QoS parameters (such as jitter, delay, capacity, in order delivery, etc). The operation returns a handle to the flow - the portId - that can be used afterwards to access the flow.
- *void write(portId, sdu)*. Sends a Service Data Unit (SDU) through the flow identified by portId. An SDU consists of user data.
- *sdu read(portId)*. Read an SDU from the flow identified by portId.
- *void deallocate(portId)*. Deallocate the flow identified by portId and release all the resources associated to it.
- *void registerApplication(appName, List<DIFName>)*. Register the application identified by appName to the DIFs specified by the list of DIF names. This operation advertises the application within a DIF, so that flows can be allocated to it. It will always be up to the application to take the final decision on refusing or accepting incoming flow allocation requests.
- *void unregisterApplication(appName, List<DIFName>)*. Unregister the application appName from all the DIFs in the specified list, or from all the DIFs (if the second argument is empty).

B. A bootstrapping example

Assume an application process *a* running on host *A* wants to communicate with an application process *b* that is registered with DIF *D* and is running on host *B* (Figure 2a). Assume also



(a) Starting situation (b) Da is instantiated

(c) Da enrolls in D through S (d) Flow is allocated from a to b

Fig. 2. Setting up communication

that there is a physical medium between *A* and *B*, with a shim DIF *S* over this physical medium. *a* will ask a management agent to allocate a flow to *b* through the IPC API. The management agent knows that *b* is reachable through *D*. If there is currently no IPC process running on host *A* that is a member of *D*, an IPC process *Da* on host *A* will need to join the DIF *D*. If necessary, the IPC process *Da* will be instantiated (Figure 2b). In order to join the DIF, *Da* has to enroll with another IPC process - reachable through the lower level DIF *S* - that is already a member of *D* (Figure 2c). *Da* knows the IPC process *Db* to enroll with and its reachability through shim DIF *S* through a management agent, and sets

up a flow with it. By means of the enrollment, *Da* obtains the DIF *D* parameters, such as the address length, or the available QoS-cubes. Depending on the DIF security policies, the enrollment procedure may include an authentication step. *a* can now allocate a flow to *b* (Figure 2d). An IPC process can also instantiate a new DIF if it is the very first IPC process to become a member of that DIF.

III. IRATI: Investigating RINA as an Alternative to TCP/IP

Prototyping of RINA started in 2010, with the main objective to verify and improve the specification drafts. There are currently three prototypes [11] [12] [13] in different degrees of maturity, all of them completely implemented in user-space. In contrast, IRATI [14] targets GNU/Linux OS platforms

and followed the fast/slow path design common in routers. Software components were placed depending on their timing requirements: components with stringent timing requirements were put in kernel space (fast path), while components with loose timing requirements were put in user space (slow path). The transport related functions were placed in kernel-space in order to allow for optimal performance while handling transport of Protocol Data Units (PDUs) and to support multiple technologies (e.g. Ethernet, WiFi, USB, FireWire). In essence, this means all the shim IPC processes, as well as all components in a normal IPC process that are related to transport were put in kernel space while all the components related to layer management - the IPC Process and IPC Manager[1] daemons - were put in user-space. Figure 3 shows how the stack is organized.

Appropriate communication mechanisms between user and kernel space had to be selected. For those operations in the fast path that are user originated, such as reading/writing of SDUs, system calls are used. For configuration and management operations, mostly residing in the slow path, netlink sockets [15] are used. These operations can be used for uni-, multi- and broadcast user/kernel originated communications.

In order to conveniently wrap all the communications between kernel and user space, a C/C++ commodity library (librina) was introduced. This library is the aggregation point for all the RINA related facilities in the system and it currently constitutes the framework that RINA applications and daemons can leverage to make use of the IRATI stack. Through the use of SWIG [16], it has been possible to generate the Java language bindings for the librina services. Therefore, the current IRATI framework enables wider adoption through the support of different languages (i.e. C, C++ and Java). The wrapping costs are minimal. Introducing bindings for other languages (e.g. Python) is expected to be straightforward. Figure 4 shows the current user-space high-level architecture.

When a user-space application needs to send an SDU, it hands it over to the kernel, where it is managed by the Kernel IPC Manager (KIPCM). In order to avoid stack overflows

[1]The IPC Manager is responsible for the creation/destruction of IPC processes

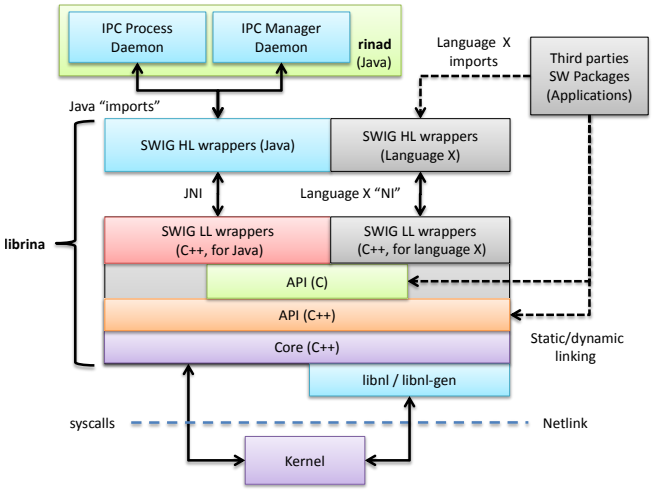


Fig. 4. IRATI Stack userspace high level architecture.

in kernel space, the recursion of RINA is transformed into iteration. The KIPCM delivers the SDU to a normal IPC process - the one assigned to support the flow to which the SDU belongs. Inside the normal IPC process, the SDU is first processed by the Error and Flow Control Protocol (EFCP) component, which groups together the data transfer parts of RINA. The EFCP turns the SDU into a full fledged PDU by prepending Protocol Control Information (PCI) for the receiving IPC process to interpret. When the EFCP is done, the PDU is passed to the Relaying and Multiplexing Task (RMT) component associated with the IPC process. By consulting its PDU Forwarding Table, the RMT decides where the PDU goes next, e.g. what port-id can be used to deliver it to the destination. At this point, the PDU is handled - interpreted as an SDU - by an IPC process in the layer below, repeating the whole process. This iterative process stops when the SDU reaches a shim IPC process, in which the forwarding is implemented using existing communication mechanisms.

The reverse path is followed upon receiving an SDU in a shim IPC process (for instance when an Ethernet frame arrives from the wire). The SDU is passed - interpreted as a PDU - to the RMT component of an higher layer IPC process (the user of the shim IPC process transport services). If the destination of the PDU is set to the address of the IPC process, the RMT hands the PDU to the EFCP instance. Otherwise the PDU is forwarded through a lower layer DIF, after consulting the PDU Forwarding Table. In the first case, the EFCP instance extracts the SDU from the PDU (removing the PCI) and gives it to the KIPCM, which passes the SDU either to a user-space application or to a higher layer DIF (in kernel-space), in which case the receiving processing starts again.

Even though IRATI's kernel-space components have been designed with performance in mind, some trade-offs were required in the first prototype in order to achieve the research objectives in a timely manner, most importantly achieving a first working proof-of-concept. Some of the trade-offs are:

- The user-space components (i.e. the IPC Process daemon,

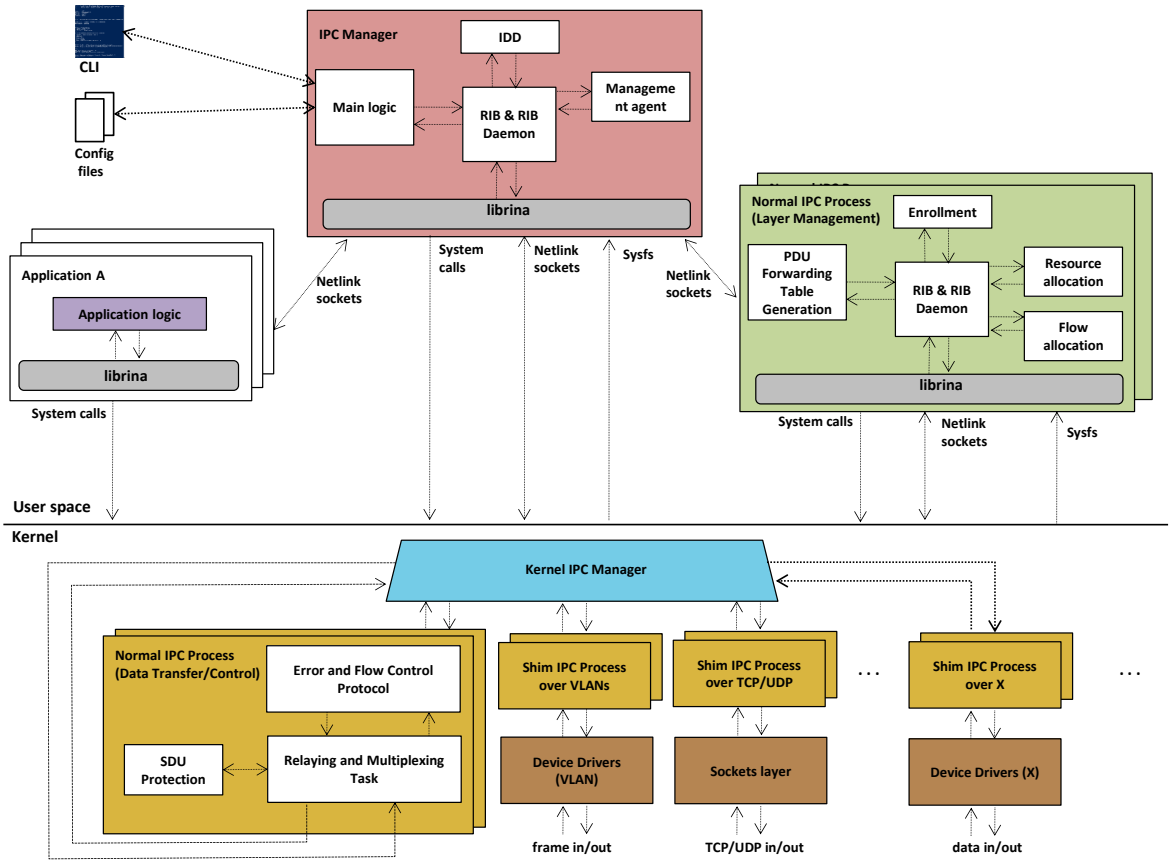


Fig. 3. The IRATI prototype components placement. [10]

the IPC Manager daemon and the RINAPerf application) were adopted from a previous prototype and are Java-based. They completely rely on the automatically generated SWIG wrappers which binds them to librina (C++).

- General optimizations are not yet in place, e.g. buffer copies (at both user- and kernel-space) have not been reduced to the bare minimum in order to increase readability of the code and facilitate debugging until the functionality is proven stable.

IV. Experiment Scenarios

The experiment depicted in Figure 5 serves two functions. First of all, it tests the functionality of the shim IPC process for 802.1Q, running over Ethernet augmented with an IEEE 802.1Q VLAN header[2] [17]. The second goal is to verify whether the normal DIF implementation supports enrollment, can establish unreliable flows[3], and allows the stacking of multiple DIFs.

The RINAperf client/server application is a RINA-native performance-measurement tool, measuring the available goodput between two application processes. The client allocates a flow with the server and the two exchange data using RINA's IPC services. The RINAperf client takes the following parameters:

- sdu-size: the SDU size to use for the test
- timeout: time interval (in milliseconds) during which the goodput between the applications is measured

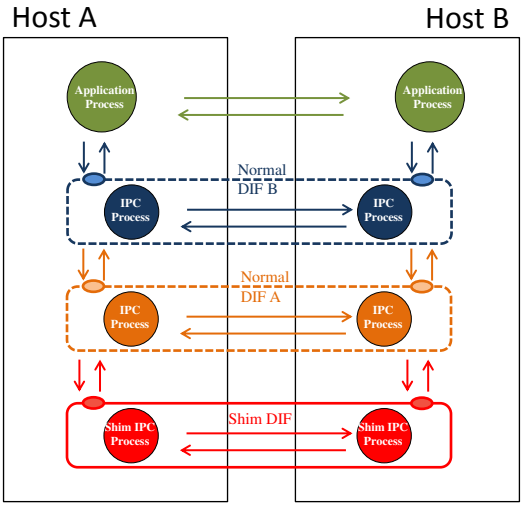
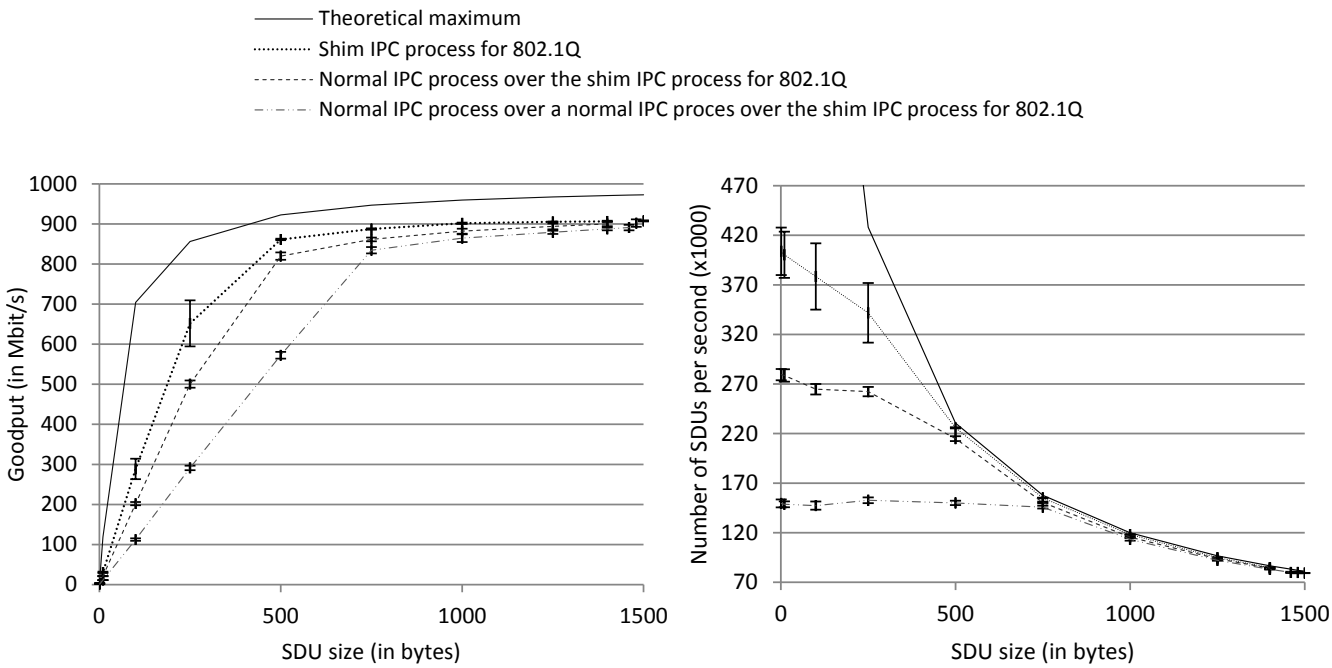


Fig. 5. The three scenarios of the use case.

[2]VLANs are a natural choice as they allow a certain level of traffic isolation on an Ethernet network, which allows identification of DIFs by VLAN id.

[3]Unreliable flows are comparable to UDP flows. Reliable flows support - comparable to TCP flows - for the IRATI stack is being designed and implemented



(a) Application goodput in function of SDU size

(b) Number of SDUs sent in function of SDU size

Fig. 6. Experiment results

Once the flow between client and server is allocated, the client tries to send as much SDUs as possible to its server counterpart. The server starts a timer with the specified timeout after it receives the first SDU. Once the timer expires, the server reports the number of received SDUs back to the client, which signals the end of the test.

We performed goodput measurements in the following 3 scenarios (also shown in Figure 5):

- The first scenario (Shim DIF + App) runs the RINAperf application directly over the shim IPC process for 802.1Q. This scenario should be the fastest, but offers the least functionality.
- The second scenario (Shim DIF + Normal DIF A + App) builds upon the first one by stacking a normal DIF on top of the shim IPC process for 802.1Q. This is a scenario that would be typically found in LANs, where the scope is the network. A lot of functionality becomes immediately available since it is inherent in the architecture: multihoming, QoS, security ...
  Note that this configuration is similar to the current Internet.
- The third scenario (Shim DIF + Normal DIF A + Normal DIF B + App) stacks another normal DIF on top. This scenario would be used in an internetwork: connecting together different networks. This scenario is added to show the influence of stacking multiple DIFs on top of each other.

V. Results

We performed experiments on 2 nodes from the iMinds OFELIA [18] island iLab.t Virtual Wall aggregate, which is a generic testbed running the Emulab software [19]. In the experiments the two nodes were connected with each other through a non-blocking 1.5 Tb/s VLAN Ethernet switch (Force 10 E1200). Each node has a Supermicro X8DTT motherboard with two Intel Xeon E5520 quad core processors, 12 GB RAM, a 160 GB HD and 6 Intel 82575EB-based GbE NICs. We used the RINAperf application to measure the maximum achievable goodput between two application processes given a certain SDU size.

In all our experiments, we set the RINAperf timeout to ten seconds and each one is repeated for different SDU sizes, ranging from 1 byte to the maximum SDU size, which is 1500 bytes when using the Shim DIF only (Ethernet). For each normal DIF we chose the address, connection endpoint id, and QoS id length to be 2 bytes, and the sequence number length to be 4 bytes. These lengths are configurable per DIF, and are chosen depending on the DIF purpose. This means that in our case, 19 bytes of PCI information is added per normal DIF. Therefore we chose the maximum SDU sizes to be 1480 bytes and 1460 bytes for scenarios two and three respectively.

The values represent the mean of the goodputs obtained, together with their respective 95 percent confidence intervals. The obtained goodput when performing these experiments can be seen in Figure 6a.

First of all, note that in GbE, the theoretical limit for the number of VLAN-tagged frames that can be sent at the maximum MTU of 1500 bytes (Ethernet physical frame size of 1,542 bytes including the inter-frame gap of 12 bytes) is 81,063 frames per second. Given the maximum SDU sizes, this puts the theoretical limit for the goodput at 972.7 Mbit/s.

As can be seen from Figure 6a, the goodput increases as

the SDU size increases, which is of course to be expected as the capacity and processing overhead due to the PCI headers decreases. The maximum throughput on the link when measured with iperf [20] is 970 Mbit/s. Adding additional normal DIFs decreases the goodput, because of the extra processing overhead and decreased maximum packet size. At the maximum MTU allowed by the system, the mean goodput achieved was 907.67 ± 1.45 Mbit/s for scenario 1, 902.09 ± 9.32 Mbit/s for scenario 2, and 891.05 ± 6.91 Mbit/s for scenario 3. So each additional normal DIF incurs a small performance penalty, but the overall goodput achieved remains close to line rate.

For small SDU sizes on the other hand, the overall goodput is not close to line rate. In order to explain the cause of this, we investigated the number of packets that were sent. Figure 6b shows the total number of SDUs sent per second in function of the SDU size.

At minimum SDU size, the number of SDUs sent per second is 403884.2 ± 23929 for scenario 1, 279482.6 ± 5685 for scenario 2, and 149554.6 ± 3999 for scenario 3. The theoretical maximum at minimum SDU size is 1488095 SDUs per second. At maximum SDU size, the number of packets sent per second is 79314.1 ± 127 for scenario 1, 79891.7 ± 826 for scenario 2, and 79994.8 ± 620 for scenario 3. The theoretical maximum at maximum SDU size is 81063 SDUs per second.

We can clearly see that, as more DIFs are stacked, the total number of SDUs per second becomes smaller, since more processing is needed per packet. For small SDU sizes, these numbers are far below the theoretical limit. There are three main bottlenecks:

- RINAPerf is written in Java and uses SWIG to request IPC services. The application is limited by how fast the operation to write a SDU is. Unlike in C++, it is not just a syscall, but it uses the Java Native Interface. Furthermore, it is single threaded, unlike iperf.
- The kernel space is not yet optimized. For instance, buffer copies are not reduced to the minimum. The stack also uses a lot of the system resources, since a lot of dynamic memory is allocated.
- No buffering is performed in the stack, except in the Ethernet driver, placing heavy strain on the kernel.

VI. Conclusions

In this paper we presented and evaluated the first performance-oriented implementation of the Recursive InterNetwork Architecture, the IRATI stack. After briefly discussing RINA, from a general perspective and through a bootstrapping example, we gave a quick overview of the design principles according to which the stack is developed. We presented a couple of scenarios which we used to demonstrate the operation and evaluate the performance. The evaluation shows that the stack can deploy multiple stacked DIFs and that stacking DIFs is quite scalable.

Performance is acceptable but the stack requires further optimization. We plan to port iperf to the RINA API, to reduce dependency on Java in user space, in order to be able to reach the theoretical limits and make a fair comparison with TCP/IP. Using SWIG to reuse components written in Java to ease application development was a correct design decision considering the time constraints, but is unsuitable for high-performance applications and performance tests, even on a high-end server.

The IRATI stack prototype is under active development. Its source code will be released as open source software, available for download from [14] in Q3/Q4 2014.

Acknowledgment

This work is partly funded by the European Commission through the IRATI project (Grant 317814), part of the Future Internet Research and Experimentation (FIRE) objective of the Seventh Framework Programme (FP7).

References

[1] C. E. Perkins, "Mobile IP," *Communications Magazine, IEEE*, vol. 35, no. 5, pp. 84–99, 1997.
[2] D. Farinacci, V. Fuller, and D. Lewis, "Lisp mobility architecture," IETF, draft-meyer-lisp-mn-01, Tech. Rep., 2010.
[3] D. Liu and H. Deng, "Mobility Support in Software Defined Networking," IETF, draft-liu-sdn-mobility-00, Tech. Rep., 2013.
[4] V. Ishakian, J. Akinwumi, and I. Matta, "On the Cost of Supporting Multihoming and Mobility," Boston University Computer Science Department, Tech. Rep., 2009.
[5] E. Trouva, E. Grasa, J. Day, I. Matta, L. T. Chitkushev, S. Bunch, M. P. de Leon, P. Phelan, and X. Hesselbach-Serra, "Transport over heterogeneous networks using the RINA architecture," in *Wired/Wireless Internet Communications*. Springer, 2011, pp. 297–308.
[6] E. Trouva, E. Grasa, J. Day, I. Matta, L. T. Chitkushev, P. Phelan, M. P. de Leon, and S. Bunch, "Is the Internet an unfinished demo? Meet RINA!" 2010.
[7] (2014, Apr.) Document Stats – What is Going on in the IETF? [Online]. Available: http://www.arkko.com/tools/rfcstats/pubdistr.html
[8] J. Day, *Patterns in network architecture: A return to fundamentals*. Pearson Education, 2007.
[9] J. Day, I. Matta, and K. Mattar, ""Networking is IPC": A Guiding Principle to a Better Internet," in *Proceedings of the 2008 ACM CoNEXT Conference*, 2008.
[10] S. Vrijders, D. Staessens, D. Colle, F. Salvestrini, E. Grasa, M. Tarzan, and L. Bergesio, "Prototyping the Recursive InterNet Architecture: The IRATI project approach," *IEEE Network*, March 2014.
[11] E. Grasa, E. Trouva, S. Bunch, P. DeWolf, and J. Day, "Developing a RINA prototype over UDP/IP using TINOS," in *Proceedings of the 7th International Conference on Future Internet Technologies*, 2012, pp. 31–36.
[12] (2014, Apr.) TRIA Network Systems, LLC. [Online]. Available: http://www.trianetworksystems.com/
[13] Y. Wang, F. Esposito, I. Matta, and J. Day, "Recursive InterNetworking Architecture (RINA) Boston University Prototype Programming Manual (version 1.0)."
[14] (2014, Apr.) The IRATI website. [Online]. Available: http://www.irati.eu
[15] P. Neira-Ayuso, R. M. Gasca, and L. Lefevre, "Communicating between the kernel and user-space in Linux using Netlink sockets," *Software: Practice and Experience*, vol. 40, no. 9, pp. 797–810, 2010.
[16] D. M. Beazley *et al.*, "SWIG: An easy to use tool for integrating scripting languages with C and C++," in *Proceedings of the 4th USENIX Tcl/Tk workshop*, 1996, pp. 129–139.
[17] S. Vrijders, E. Trouva, J. Day, E. Grasa, D. Staessens, D. Colle, M. Pickavet, and L. Chitkushev, "Unreliable inter process communication in Ethernet: migrating to RINA with the shim DIF," in *5th International Workshop on Reliable Networks Design and Modeling (RNDM-2013)*, 2013, pp. 97–102.
[18] (2014, Apr.) The OFELIA website. [Online]. Available: http://www.fp7-ofelia.eu
[19] (2014, Apr.) The Emulab website. [Online]. Available: http://emulab.net/
[20] (2014, Apr.) iperf. [Online]. Available: http://code.google.com/p/iperf/