A Software Development Kit to exploit RINA programmability

Vincenzo Maffione, Francesco Salvestrini
Nextworks
Pisa, Italy
Email: {v.maffione, f.salvestrini}@nextworks.it

Eduard Grasa, Leonardo Bergesio, Miquel Tarzan
Fundacio I2CAT
Barcelona, Spain
Email: {eduard.grasa, leonardo.bergesio, miquel.tarzan}@i2cat.net

***Abstract*—The Recursive InterNetwork Architecture (RINA) is a general architecture for all forms of computer networking, based on a single type of programmable layer that recurs as many times as required by the network designer. The recursion and programmability aspects of RINA are key to design flexible, heterogeneous networks while still bounding their complexity. In this paper we show how the programmability enabled by the RINA architecture can be exploited in practice by means of a Software Development Kit (SDK) developed for IRATI, the open source RINA implementation. A proof of concept validation of the SDK is carried out by experimenting with multiple policies in a distributed cloud network scenario.**

I. Introduction and Motivation

Today's networks have to cope with a great variety of electronic devices - ranging from tiny, resource-constrained sensors to high-performance machines in datacentres, including fast-moving mobile devices - and with the evolution of distributed applications and protocols with their ever-increasing differentiated requirements. Since networks are required to be flexible enough to adapt to different situations, a catch-all strategy for network design and administration is not achievable. At the same time, complexity should be bounded in order to limit the cost of management and security [1]. To cope with differentiated requirements, the Internet community constantly introduces new protocols, resulting into networks becoming more and more complex to manage. Looking at the IETF RFCs statistics [2], it can be seen that the number of new RFCs per year is constantly increasing with no evidence of a possible slowdown.

The Recursive InterNetwork Architecture (RINA) has recently been proposed [3], [4] as a fundamental approach to networking, to mitigate or overcome well-known limitations of the current TCP/IP-based Internet architecture. RINA is founded on the concept of networking as Inter Process Communication (IPC). Whereas in the Internet architecture each layer contains a different set of functionalities and offers different APIs to the upper layers, RINA defines a single programmable layer - known as Distributed IPC Facility (DIF) - that contains all the functions that are needed to provide IPC services to applications or higher level DIFs. Each DIF provides IPC services over a limited scope, e.g. a LAN, a campus network, an ISP network or the whole Internet. Each host participating in a DIF must run a local agent for that DIF, known as IPC Process (IPCP). The DIF itself is therefore

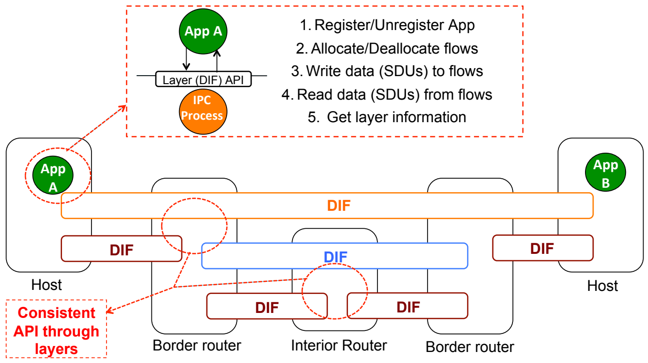
App A
1. Register/Unregister App
2. Allocate/Deallocate flows
3. Write data (SDUs) to flows
4. Read data (SDUs) from flows
5. Get layer information
Layer (DIF) API
IPC Process
App A
App B
DIF
DIF
DIF
DIF
DIF
DIF
Host
Host
Consistent API through layers
Border router
Interior Router
Border router

Fig. 1. An example of the structure of RINA

a distributed application that exposes an API to provide IPC services to its users. Recursion in RINA is used to build wider-scope DIFs on top of smaller-scope ones: a DIF at level N uses the IPC services offered by one or more DIFs at level $N-1$, as illustrated in Figure 1. DIFs at the lowest level are known as *shim* DIFs: they provide the DIF API, with limited IPC services, by directly wrapping existing legacy technologies (e.g. Ethernet, WiFi, TCP/IP, ...) rather than using lower level DIFs.

The RINA architecture specifies the different functions (or components) contained in the DIF, and a set of variable behaviours (policies) for each function. This allows the network administrator to properly select policies for each DIF depending on its scope, the environment where it operates, and the kind of service provided by its lower level DIFs.

The main contribution of this paper is the design and implementation of a Software Development Kit (SDK) to support DIF programmability on top of the IRATI open-source RINA prototype [5] [6]. We believe that programmability in RINA is the key to bound network complexity and limit the cost of network administration. When new requirements arise for a network, the designer can select or write new policies for the involved DIFs, without the need of introducing new protocols. The SDK allows DIF policies to be dynamically replaced while the network is operating, without causing IPC service disruption.

In the rest of this paper, section II contains an overview of the advantages of RINA in terms of virtualization and programmability, and a comparison with related work. Section

III reports our main contribution, the design of a SDK for the IRATI prototype. Section IV provides a Proof of Concept evaluation of the SDK. Finally, section V contains our conclusions and future work.

II. The RINA alternative approach: comparison to related work

Results analyzing the advantages of RINA over other approaches to network architecture have been already published in scientific forums. In [7] Ishakian et al. show that RINA supports multihoming and mobility with less cost and more performance than mobile IP [8] and LISP [9]. Trouva et al. [10] perform an initial analysis on why RINA provides a better framework than the current Internet for supporting transport over heterogeneous networks. In [11] Trouva et al. discuss how applications can dynamically be discovered across layers and how new layers can be created on the fly. Bodappati et al. [12] shows that because of the decoupling of transport port allocation and access control from data synchronization and transfer, RINA is much more resilient than TCP/IP to transport-level attacks such as port-scanning, connection opening or data-transfer. Van der Meer at al. [13] highlight that the commonality and autonomic properties exhibited by RINA layers provide an ideal environment to explore automated and responsive network management approaches.

Since this paper is focused on the recursion and programmability aspects of RINA, we motivate our contribution by providing an initial comparison of RINA with related work on these areas.

A. Recursion and virtualization

Current networks are based on a layered architecture where each layer has a different function (physical, data link, network, transport). Layers are used as units of modularity and functional decomposition rather than as a tool to isolate different scopes. As described in the next paragraphs this model does not work in practice and requires extensions (such as "virtual layers" or "layers 2.5") to accommodate the required differences in scope found in real networks.

IEEE 802.1Q (VLANs) allows to multiplex up to 4094 independent broadcast domains over the same Ethernet network. The scope of each VLAN domain is the whole Ethernet network or part of it. VLAN segmentation run early into scalability problems: the limited VLAN-id space and the desire of network providers to carry customer VLANs transparently over their networks gave birth to the IEEE 802.1ad [14] (Q-in-Q) standard, introducing another layer to multiplex several customer VLANs transparently into a single provider VLAN. IEEE 802.1ah [15] (PBB) further improved the separation of customer and provider layers by including source and destination MAC addresses in the provider Ethernet layer, thus allowing the decoupling of the provider and customer routing.

Multi Protocol Label Switching (MPLS) is another example of the need for a technology to isolate different scopes, in this case to enable forwarding strategies that are independent from the one used in the global Internet network layer.

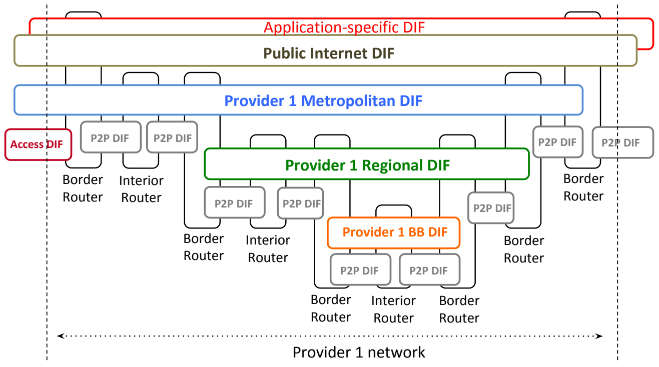

Fig. 2. Service provider network design with RINA, using DIFs with different scopes and requirements

Service providers deploy MPLS as an extra layer (known as layer 2.5) on top of the data-link layer in order to tightly control the forwarding of traffic across their networks, usually applying Traffic Engineering (TE) techniques to support the QoS requirements of their customers. MPLS defines a protocol header containing a label used by MPLS-enabled devices to decide how to forward packets. MPLS headers can be stacked (following the same pattern as Q-in-Q or PBB) enabling multiplexing of higher MPLS layers onto lower MPLS layers. Label stacking can be used to support L2 or L3 Virtual Private Networks (VPN) [16], where the provider infrastructure supports independent scopes dedicated to different customers.

Datacentre (DC) networks also require isolating different scopes, specially to support multi-tenancy [17]. Multi-tenancy allows a DC provider to partition its infrastructure into independent networking domains dedicated to different customers and/or applications. It is desirable that functions at each domain are adaptable to the customer requirements. A way to achieve multi-tenancy would be to use the protocols and techniques described in the previous paragraphs (VLANs, PBB, MPLS). However, since these protocols have been designed for network service provider environments, other approaches better suited to some DC environments have been developed under the umbrella of the "overlay virtual networks" concept. The main idea is to introduce per-tenant "virtual network stacks" (L2 to L4), usually with their dedicated control plane. These extra layers are implemented on the DC servers and overlaid on top of the DC fabric's transport layer using a tunneling protocol such as VXLAN, STT or NVGRE [18].

These examples show that while traditional network architecture starts from a limited set of layers with different functions, it ends up with a wider set of layers with repeated functions that are used to isolate different scopes.

In contrast, RINA can support differences in scope in a natural way, without introducing new concepts (such as network virtualization), types of layers nor protocols. Since the architecture provides a variable number of programmable layers the network designer can use them to create customized, independent networking environments. The functions in each layer can be customized to the layer's operational environment, as described in section II-B. Figure 2 illustrates a potential design for an ISP, with a number of internal DIFs supporting public Internet and VPN services for its customers. The ISP uses three internal layers to aggregate and transport the traffic between its points of presence: the metropolitan DIF brings together all the metropolitan networks which connect to customers or other ISPs; the regional DIF aggregates the traffic of the metropolitan areas and provides inter-metro connectivity; while the backbone DIF aggregates the traffic of the regional areas and provides inter-region connectivity.

The recursive nature of RINA, therefore, allows to use the DIF basic block at different scopes and with different policies. By reusing the same DIF API and mechanisms there is no need to introduce new protocols. New layers can be added dynamically and without requiring additional hardware/software support.

B. Programmability

Work on adaptive, programmable networks capable of accommodating to different operational environments and to support ever-changing application requirements has been carried out for more than 20 years [19]. In [20] Campbell argues that programmable networks are key to the rapid creation and deployment of new network services, and surveys the programmable network proposals of the time, which can be roughly divided between the open signaling (OPENSIG) and active networks approaches. OPENSIG argued for a set of open, programmable network interfaces that provided external programs access to the internal state and control of network devices. Active networks advocated the dynamic deployment of network services at runtime, via special packets containing executable code and similar mechanisms.

The OPENSIG efforts crystalized on standards such as i) the IETF General Switch Management Protocol (GSMP) [21], which enabled the partitioning of a label switch (ATM, Ethernet) into multiple ”virtual switches”, allowing external controllers to manage the ”virtual switches” via the GSMP protocol; ii) the IETF FORCES initiative [22], which defines a framework and associated protocols to standardize the information exchange between the control and forwarding planes of an IP network element; and iii) the IEEE P1520 [23] standard project, which aimed at establishing a reference model for networks APIs. Although these technologies saw some deployments, they never gained the traction that the current Software Defined Networking (SDN) trend has in the networking industry and research communities. SDN can be seen as a re-incarnation of OPENSIG, which initially used the OpenFlow protocol as a means to control the forwarding of TCP, UDP, IP and Ethernet packets through network nodes from an external controller (in a similar way to GSMP).

A number of limitations have already been identified for the first wave of SDN technologies. First, the flexibility of OpenFlow as the protocol between the controller and the device being controlled: since SDN does not specify a network architecture, the controller-device protocol should be able to define rules on arbitrary fields in order to be evolvable. Flexible interfaces to define mechanisms for parsing packets and

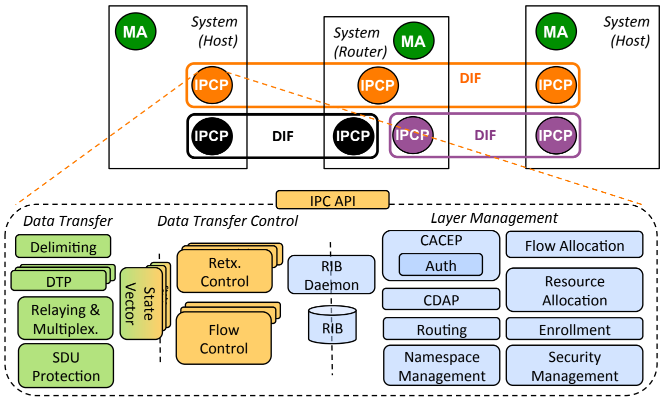

Fig. 3. The programmable functions of a layer

matching header fields such as P4 [24] are being researched to address this issue. Another problem is the scalability and resiliency of centralized controllers; approaches to distributed SDN control in order to improve resiliency [25] and allow for per-domain controllers [26] are being investigated. Recursive SDN controllers are also foreseen to allow for the partitioning and control of network resources at more granular scopes [27].

RINA takes a different approach to network programmability, leveraging the fact that it is a working hypothesis for a general theory of computer networking. The functions performed by each layer, illustrated in Figure 3, can be divided in three categories of growing timescale and complexity [4]: data transfer (forwarding/sending/receiving PDUs), data transfer control (flow and retransmission control) and layer management (enrollment of new IPCPs, routing, namespace management, flow allocation, resource allocation, IPCP authentication, application access control, security). RINA uses the principle of separation of mechanism and policy to support programmability. Mechanism is the invariant, fixed part of DIF functions, while policy is the variable, programmable behavior of the functions that can be adapted to the particular scenario where the DIF is operating. For example, all layer management functions use the same protocol (CDAP, the Common Distributed Application Protocol) to exchange information with their peers, but the objects being exchanged using the protocol can vary. As another example, all DIFs have the same mechanism for packet forwarding, but the packet scheduling policy can be programmed.

Implementing new policies is the way to cope with new or unexpected requirements, rather than designing and implementing new full-fledged protocols. As reported in section III, hot-replacement of policies is possible, and does not cause IPC service disruption (e.g. deallocation of application flows).

Since in RINA all layers reuse the same extensible data transfer protocol (EFCP, the Error and Flow Control Protocol) [28] there is no need to define mechanisms for generic header matching like in [24]. Moreover, due to the distributed nature of its layer, RINA has the potential to mitigate the resiliency and scalability issues of centralized layer management approaches, since responsibilites can be splitted across all the IPCPs in a DIF. Centralized solutions are still possible, since

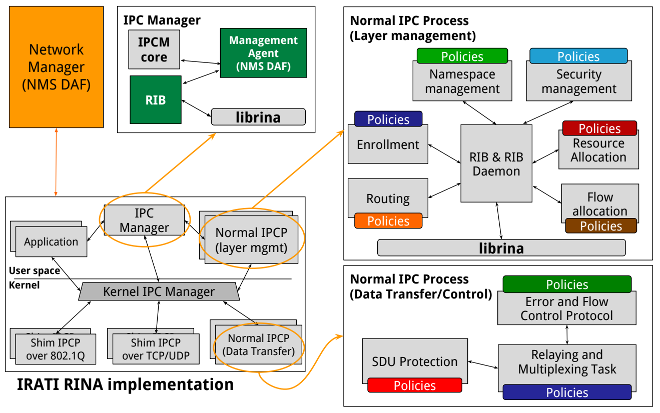

Fig. 4. Software components of the IRATI implementation and their policies supported by the SDK

RINA nodes can export the local IPCP RIBs to a remote entity, e.g. a central Manager. Layer management could therefore be implemented by the Manager remotely accessing the RIBs through the CDAP protocol.

In contrast with SDN, programmability of the layer functions is not limited to packet forwarding; policies can be implemented for transmission control, flow control, resource scheduling, multiplexing, routing, authentication, encryption, etc, as reported in section III.

III. Design and implementation of an SDK to make the IRATI implementation programmable

After a brief introduction to IRATI RINA implementation, this section presents our contribution: a SDK to support RINA programmability in IRATI (illustrated in Figure 4).

A. Introduction to the IRATI RINA implementation

The IRATI RINA open-source implementation is the main outcome of the FP7-IRATI project, targeting Linux-based operating systems. In IRATI, the implementation of DIF functions is splitted between user-space and kernel-space. Data transfer and device low-level access functionalities are implemented in kernel-space, while layer management functionalities are implemented in user-space. The software is organized in four packages:

- Modified version of the 4.1.16 Linux kernel, including system call support for managing IPCP and packet I/O, data transfer functionalities and shim DIFs (see section I).
- *librina*, the main library to be used by applications to access the DIF API (flow allocation, application registration, packet I/O).
- *rinad*, containing the implementation of userspace parts of the DIF functions, i.e. the layer management functions.
- *rina-tools*, containing testing tools to generate or receive RINA traffic.

The IRATI stack uses different daemons on the hosts where it runs. An IPCP daemon is run for each DIF the host is part of. This daemon represents an IPCP and implements the managment functions of the DIF. In addition to the IPCP daemons, a single IPC Manager (IPCM) daemon is in charge of i) controlling the life-cycle of all the IPCPs on the host, and ii) acting as a message broker between applications, kernel and IPCP daemons. These messages are used to implement the layer management functions, like the flow allocation procedure which involves both applications and IPCPs. Applications, instead, send and receive data packets using system calls, without the need to interact with the IPCM.

B. SDK requirements

RINA defines a clear separation between mechanism and policy, as explained in section II-B. The RINA specifications define the different policy points where the IPCP behaviour can be programmed, together with a default behaviour. As an example, a PDU scheduling policy is defined in the Relay and Multiplexing Task (RMT) component of the IPCP. Each time the RMT wants to dequeue a PDU from its input or output queues, this policy is invoked to decide what queue should be served first. Each internal component of the IPCP has a well defined set of policy points that may be tweaked to adapt the component's behaviour.

Each IPCP running in a host can be programmed independently. This is a fundamental requirement, since different IPCPs in the same host (typically) belong to different DIFs, and so need different behaviours to adapt to different scopes and enviroments.

In addition to the default one, several policies can be defined on the same policy point. This is an important requirement, since the needs of a specific IPCP may change over time, and therefore an on-line replacement may be convenient.

The code of a policy may need to interact with the data model of the associated component to get information or update its data structures. In IRATI, the points where policies are invoked can correspond to either user-space or kernel-space code, depending on the related IPCP component. The scheduling policy of the RMT, as an example, is part of the kernel-space code. The SDK must provide mechanisms to "plug-in" (i.e. dynamically load) either user-space and kernel-space policy code at run-time, and to replace policies for components while their are running (e.g. processing PDUs).

Policies are not necessarily independent of each other, but may interact or cooperate, e.g. there may be some state shared by different policies. The possible interactions depend on algorithms and implementations. As an example, an RMT scheduling policy implementing priority based on a traffic classification scheme may have to cooperate with the RMT Queue Monitoring policy (invoked when PDUs are enqueued/dequeued) in order to keep the shared data structures required by the priority algorithm. In other words, a plugin may need cooperating policies in order to implement a higher level strategy, e.g. a priority based packet classifier and scheduler.

C. SDK design

With these requirements in mind, we designed and integrated into IRATI a SDK that allows for dynamic loading of policy plugins and hot-replacing of policies.

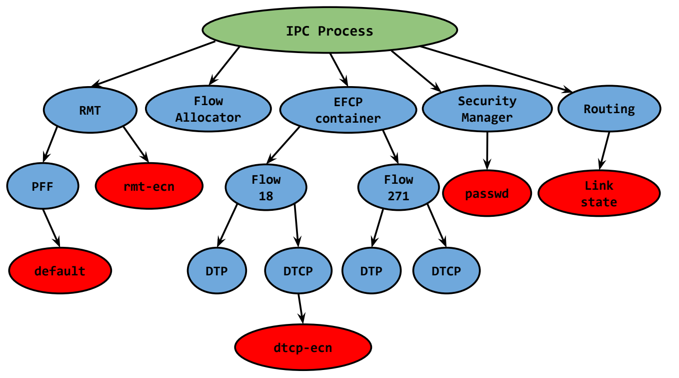

Fig. 5. A simplified version of the IPCP components tree model. The root represents the IPCP itself, blue nodes are the IPCP components, and red nodes are policy-sets.

To allow cooperation among policies, the *policy-set* concept was introduced. A policy-set defines a group of different policy points, as defined in the RINA architecture, that may want to cooperate with each other. After an analysis of the defined policy points and the expected interactions, we decided to group them on a per-component basis, so that each component of the IPCP has its policy-set. The reason of this choice is that most of the interactions happen between policies belonging to the same component. Possible interactions between policies of different components can still be handled through interactions between components as allowed by the IRATI internal APIs.

The policy-set is also the atomic unit that the SDK can load and instantiate on components. A plugin may contain one or more policy-sets - for the same or different components. At plugin loading time, the policy sets contained in it are published (loaded) into the SDK, and made available for later instantiation on running components. Unloading of a plugin can happen only when there are no more running instances of any of its policy-sets. Therefore, a per-plugin reference counting mechanism was necessary.

A policy-set instance holds a reference to its associated component instance, to interact with it using a per-component ad-hoc API. These APIs allow policy code to access the component data model in a controlled way: only the required functionality is exposed, while critical data structures can be hidden.

To enable policy-set instantiation and replacement, all the IPCP components have been extended to accept or propagate a replacement configuration message. For this purpose, we modeled the IPCP structure as a tree of components with the IPCP itself at its root, as illustrated in Figure 5. Given two components X and Y, X is a child of Y if and only if X is a subcomponent of Y. As an example, the PDU Forwarding Function (PFF) component is a child of the RMT. Moreover, also policy-sets are modeled as children of the components they are associated to. The configuration messages are issued by the IPCM daemon and sent to the IPCP to be configured through the IRATI netlink infrastructure. A message contains the name of the replacing policy-sets and a string identifying the specific component to address. The identifier specifies the uinque path in the tree that connects the component from the root, in dotted notation. The message is forwarded by each IPCP component in the path, following the order specified by the identifier. The addressed component (the last in the path) will not forward but accept the message, instantiate the specified policy set and replace the current one. This hierarchical policy configuration approach has been designed to be extensible and modular.

Finally, some policies may want to expose implementation-specific tunable parameters to the DIF administrator. As an example, an RMT scheduling policy may have some algorithm-specific thresholds that can be adjusted to tune performance. The SDK supports modification of such parameters with an additional configuration message that is delivered to the involved policy-set using the same hierarchical delivery mechanism described above.

D. *User-space plugin infrastructure*

User-space plugins are implemented as shared objects (i.e. dynamic libraries) dynamically loaded by the IPCP daemons using the *libdl* library. Since IRATI user-space code is written in C++, we used Object Oriented (OO) techniques to implement SDK support. Each policy-set is represented by an abstract class whose interface contains the policy methods specified for the associated IPCP component. Plugins provide implementation for policy-sets by deriving concrete classes from the abstract ones. Abstract classes contain a reference to their associated IPCP component, so that the concrete implementation can access the ad-hoc component API.

The *abstract factory* design pattern is used to instantiate and destroy policy-set classes [1]. When an IPCP daemon loads a plugin, it obtains policy-set factories for all the published policy-set classes.

E. *Kernel-space plugin infrastructure*

Kernel-space plugins are implemented as Loadable Kernel Modules (LKMs) and loaded in the running IRATI kernel by the IPCM daemon. Although kernel-space code is written in C, we used OO techniques similar to those reported in section III-D. A policy-set is represented as a C struct containing function pointers (one for each policy point) and a reference to its associated component. Plugins provide implementations for those functions. At plugin loading time, factories for all published policy-sets are registered to the kernel-space SDK and become available to the DIF administrator. In order to support hot-replacing of policy sets with very low locking overhead for read operations, all references to policy-set instances are protected by RCU [29] locks.

F. *Policy Catalog*

The Policy Catalog is a submodule of the IPCM daemon that has been implemented to carry out coordination tasks related to the SDK, including:

- (Un)Load kernel-space plugins (LKMs)

[1]This was necessary because C++ mangled symbols (e.g. class constructors) cannot be invoked across the shared object boundary

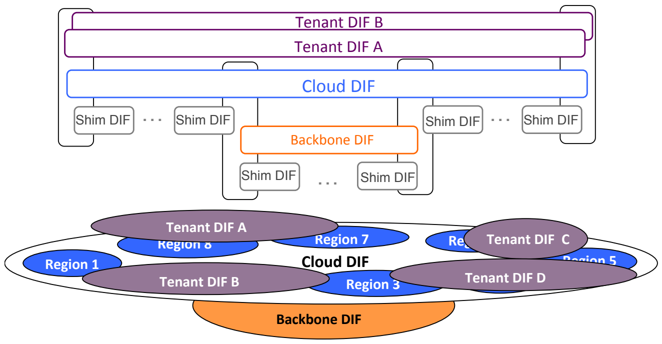

Fig. 6. Distributed cloud network layers

- Ask IPCP daemons to (un)load user-space plugins
- Issue policy-set configuration messages
- Issue messages to configure tunable parameters
- Keep track of loaded plugins and policy-sets
- Keep track of the currently used policy-sets for the components of each IPCP running in the host

IV. Proof of Concept Validation

In this section we experimentally validate the functionalites of the SDK and probe the adaptability of RINA in practice by means of an experiment that designs the network of a distributed cloud service provider using RINA. Servers providing the cloud's computing power are not located in datacentres but in people homes, offices, etc. The DIF structure for the distributed cloud network, illustrated in Figure 6, is composed of three types of DIFs: i) multiple tenant DIFs floating on top of the cloud DIF, providing a customized and isolated networking environment to each tenant; ii) a cloud DIF divided in multiple regions that brings all the distributed cloud servers in a single resource pool and iii) a Backbone DIF that supports aggregation of traffic and provides direct inter-region connectivity to the cloud DIF.

All DIFs have the same structure and protocols, but configured with different policies. Although a full discussion of the different policies used in each DIF is not included for space reasons, next subsections focus on two examples that illustrate the possibilities enabled by our SDK implementation: programming the IPCP's data transfer functions to implement congestion control strategies and programming the IPCP's layer management functions to customize the routing algorithm for a DIF.

A. Data transfer programmability: congestion control

By defining a custom policy-set for EFCP (RINA's data transfer protocol) and a custom policy-set for RMT (which is in charge of forwarding and multiplexing packets over N-1 flows), we implemented two congestion control (CC) solutions based on a combination of window-based flow-control and Explicit Congestion Notification (ECN). In both solutions all the data transfer (EFCP) connections in the DIF are flow-controlled by the EFCP receiver, regardless of the reliability of the connection. RMTs in the IPCPs between the sender and the receiver monitor their queues; if the queue size goes over a certain threshold, pakets are marked with an ECN flag. The EFCP protocol machine at the receiver IPCP checks the ECN flag of the received packets; reducing or increasing the sender's credit depending on the ECN marks. One solution adapts DEC's binary feedback scheme for congestion avoidance [30] to the DIF environment, while the other uses the Random Early Detection (RED) algorithm [31] to mark packets in the RMT and an adaptation of TCP Tahoe's congestion management algorithm in the EFCP receiver.

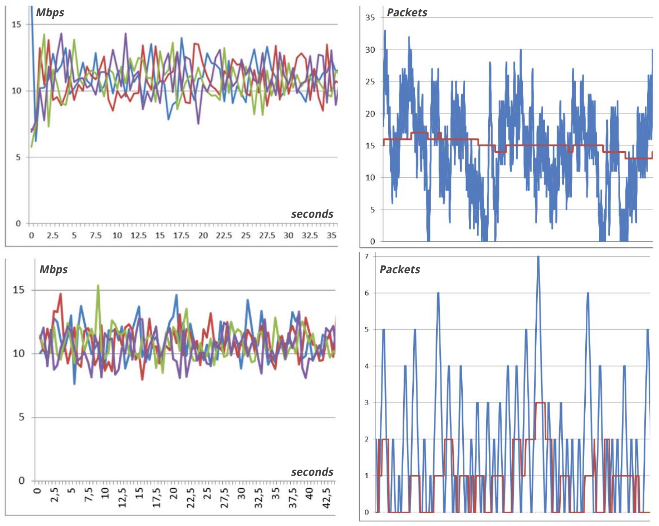

Fig. 7. On top, RED+TCP CC policies. At bottom, DEC binary feedback CC policies. On the left, throughput(Mbps) vs. time(s). On the right, RMT queue occupation(packets).

Both CC solutions can be used by cloud DIF and the backbone DIF. Figure 7 shows the results of both solutions in action when 4 parallel flows in the cloud DIF use a physical link with 50 Mbps bandwidth. The Figures show that in both cases the flows share the link bandwidth in a fair way, having a higher RMT queue occupation in the RED-TCP solution than in the DEC's one, as expected.

Each CC solution is implemented as a kernel-space plugin packaging a couple of policy-sets, using about 1000 lines of C code, including the implementation of specific RMT/EFCP functionality, factory interfaces, module initialization and cleanup.

B. Layer management programmability: routing

Routing algorithms have two main tasks: maintain a consistent distributed view of the routing information across the routing peers, and computing routes based on that information. In order to realize the first task, RINA routing policies can leverage the layer management functions: the Resource Information Base (RIB) as a distributed database and the CDAP protocol to access peers' RIBs. In this way the effort of specifying and implementing routing algorithms is considerably reduced, since developers can focus on graph algorithm and distributed database strategies, rather than on information exchange format and serialization.

We ported the IRATI link-state routing algorithm implementation to the SDK, by separating the specific link-state routing code into its own policy-set. The link-state implementation defines a set of RIB objects to model the routing information exchanged with peers: a link-state database containing multiple entries. Each entry describes a flow (i.e. a link) between two neighbor IPCPs. Once an IPCP has the complete graph of the IPCPs in the DIF, it uses a Shortest-Path algorithm to compute its routing table towards all the other nodes. Link-state database entries are exchanged by means of CDAP operations, that allow an IPCP to create, delete, or read remote RIB objects.

The link-state routing policy-set is implemented in about 1500 lines of C++ code.

V. Conclusions and Future work

When dealing with unexpected and ever-changing user requirements, RINA allows network design to transition from devising new protocols to defining new policies for a reduced set of well-defined DIF components. The SDK presented in this paper shows a practical realization of this approach, enabling the programmability of a C/C++ RINA implementation for Linux operating systems. The SDK allows network architects and administrators to configure each layer with the policies that are most appropriate to its scope and operating environment. Since the introduction of new policies is part of the RINA architecture, the SDK has the potential to reduce the involved efforts in terms of design, implementation and deployment, as compared with the introduction of new protocols.

To fully validate our SDK implementation and the policy points defined by RINA specifications, we are actively developing new plugins and setting up large experimentation scenarios with frequently switching policies.

Acknowledgment

This work is partly funded by the European Commission through the FP7 PRISTINE project (Grant 619305). The authors would like to thank all PRISTINE partners that contributed to the design and development of the SDK.

References

[1] B. Schneier, “A plea for simplicity: You can’t secure what you don’t understand,” *Information Security*, 1999.

[2] (2014, Apr.) Document Stats – What is Going on in the IETF? [Online]. Available: http://www.arkko.com/tools/rfcstats/pubdistr.html

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

[4] 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.

[5] 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.

[6] (2015, October) Irati rina implementation. [Online]. Available: https://github.com/IRATI/stack/

[7] V. Ishakian, J. Akinwumi, F. Esposito, and I. Matta, “On supporting mobility and multihoming in recursive internet architectures,” *Comput. Commun.*, vol. 35, no. 13, pp. 1561–1573, July July, 2012.

[8] J. A. C. Perkins, D. Johnson, “Mobility support in ipv6,” Internet Engineering Task Force, Tech. Rep., 2011.

[9] D. M. D. L. D. Farinacci, V. Fuller, “The locator/id separation protocol (lisp),” Internet Engineering Task Force, Tech. Rep., 2013.

[10] 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.

[11] E. Trouva, E. Grasa, J. Day, and S. Bunch, “Layer discovery in rina networks,” in *Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), 2012 IEEE 17th International Workshop on*, 2012, pp. 368–372.

[12] G. Boddapati, J. Day, I. Matta, and L. Chitkushev, “Assessing the security of a clean-slate internet architecture,” *Proceedings of the Network Protocols (ICNP), 2012 20th IEEE International Conference on*, 2012.

[13] S. van der Meer, J. Keeney, and L. Fallon, “Dynamically adaptive policies for dynamically adaptive telecommunications networks,” *Proceedings of IEEE 11th International Conference on Network and Service Management, CNSM 2015*, 2015.

[14] IEEE, “802.1ad - provider bridges,” International Standard, May 2006.

[15] ——, “802.1ah - provider backbone bridges,” International Standard, June 2008.

[16] L. Cittadini, G. Di Battista, and M. Patrignani, “Recent advances in networking, chapter 6, mpls virtual private networks,” online, http://www.sigcomm.org/content/ebook, August 2013.

[17] M. Bari, R. Boutaba, R. Esteves, L. Granville, M. Podlesny, M. Rabbani, Q. Zhang, and M. Zhani, “Data center network virtualization: A survey,” *IEEE Communications Surveys and Tutorials*, vol. 15, no. 2, 2013.

[18] (2011, Oct.) Problem Statement: Overlay for Network Virtualization. [Online]. Available: http://tools.ietf.org/html/draft-narten-nvo3-overlay-problem-statement-01

[19] A. T. Campbell, I. Katzela, K. Miki, and J. Vicente, “Open signaling for atm, internet and mobile networks (opensig’98),” *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 1, 1999.

[20] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela, “A survey of programmable networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 2, 1999.

[21] A. Doria and K. Sundell, “Rfc 3294: General switch management protocol (gsmp) applicability,” Internet Engineering Task Force, Tech. Rep., 2002.

[22] L. Yang, R. Dantu, T. Anderson, and R. Gopal, “Forwarding and control element separation (forces) framework,” Internet Engineering Task Force, Tech. Rep., 2004.

[23] J. Biswas, A. Lazar, J.-F. Huard, and K. Lim, “The ieee p1520 standards initiative for programmable network interfaces,” *IEEE Communications Magazine*, vol. 36, no. 10, October 1998.

[24] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming protocol-independent packet processors,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, July 2014.

[25] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, “A distributed and robust sdn control plane for transactional network updates,” in *INFOCOM*, 2015.

[26] K. Phemius, M. Bouet, and J. Leguay, “Disco: Distributed multi-domain sdn controllers,” in *Network Operations and Management Symposium (NOMS), 2014 IEEE*, 2014, pp. 1–4.

[27] O. N. Foundation, “Sdn architecture, issue 1,” Open Networking Foundation, TR, Technical Reference SDN ARCH 1.0 06062014, 2014.

[28] K. Mattar, I. Matta, J. Day, V. Ishakian, and G. Gursun, “Declarative transport: A customizable transport service for the future internet,” *In Proceedings of the 5th International Workshop on Networking Meets Databases (NetDB 2009)*, 2009.

[29] (2015, Oct.) Read Copy Update documentation. [Online]. Available: https://www.kernel.org/doc/Documentation/RCU/rcu.txt

[30] K. Ramakrishnan and R. Jain, “A binary feedback scheme for congestion avoidance in computer networks with connectionless network layer,” *ACM Transactions on Computer Systems*, vol. 8, no. 2, pp. 158–181, 1990.

[31] S. Floyd and V. Jacobson, “Random early detection gateways for congestion avoidance,” *IEEE/ACM Trasactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993.