Rumba: a Python Framework for Automating Large-Scale Recursive Internet Experiments on GENI and FIRE+

Sander Vrijders and Dimitri Staessens
Ghent University-imec
Technologiepark 15
9052 Zwijnaarde, Belgium
Email: {sander.vrijders, dimitri.staessens}@ugent.be

Marco Capitani and Vincenzo Maffione
Nextworks, s.r.l.
Via Livornese 1027
56122 Pisa, Italy
Email: {m.capitani, v.maffione}@nextworks.it

***Abstract*—A number of recent EU-funded projects have been investigating the Recursive Internet Architecture (RINA). IRATI built an initial prototype implementation, which was extended by the PRISTINE project towards technology demonstrators showing the feasibility of the architecture and demonstrating how RINA tackles security and reliability and how it can simplify network management. Currently, ARCFIRE sets out to evaluate realistic network scenarios, scaling up experiments in terms of numbers of nodes, services and running time. In this paper we present Rumba, a free open source experimentation framework developed within ARCFIRE in order to drastically reduce the time required to deploy and conduct such large experiments. Rumba is powerful yet easy to use. It provides a simple abstraction to model the RINA network as well as APIs for reserving testbed resources, installing the prototype, configuring and bootstrapping the recursive network, running the experiment scenario, collecting the results data and releasing the testbed resources. Rumba provides QEMU, jFed and emulab support to run experiments on a local machine or on various US and EU testbeds provided by GENI and FIRE+. Our experiences show that Rumba reduces the time required to configure and run large experiments using the RINA prototypes by several orders of magnitude.**

I. Introduction

The enormous success and rapid growth of tech companies that manage large datacenters has introduced a shift in the telecoms industry where service providers aim to adopt datacenter network management frameworks to simplify management of their network as to reduce capital and operational expenditures. To this end, networks research efforts related to Software Defined Networking (SDN) and Network Function Virtualization (NFV) [1] have introduced a shift away from traditional hardware-based *black-box* network appliances to programmable *white-box* network elements. This trend of network softwarization has led to Open Source projects such as ONOS, CORD [2] and OpenDaylight [3]. Similarly, Open Source Software is playing an increasing role in research, where various EU-funded projects are developing prototypes and frameworks that are made available to the public. Projects such as Superfluidity [4] and SONATA [5] provide different solutions for NFV network orchestration.

With a growing number of available software projects and prototypes, there is a need for deploying and evaluating all these novel technologies at scale in controlled experiments. This need has been timely addressed by the Global Environment for Network Innovation (GENI) [6] and the Future Internet Research and Experimentation (FIRE) [7] initiatives in the United States and European Union respectively, which provide virtual laboratories that can support computer network experiments with a large number of physical or virtual machines. The recent Fed4FIRE project [8] federates a large number of testbeds from FIRE and GENI and provides the jFed [9] framework to access them in a uniform way.

Other research efforts, such as the Recursive InterNetwork Architecture (RINA) [10], arrive at network programmability starting directly from a distributed computing model. The IRATI [11] project was funded through the FIRE initiative and developed a first RINA implementation for GNU/Linux systems. PRISTINE [12] extended the IRATI prototype with scalable solutions for security, seamless mobility, routing and reliability. These projects have deployed the prototypes to perform small-scale integration tests in order to validate RINA and its policies. Although small size deployments can be useful for basic functional testing, design or implementation problems often show up only when deploying the new technologies at scale. The ARCFIRE project [13] aims at experimenting with RINA at a larger scale, deploying experiments with hundreds to thousands of nodes and thousands to tens of thousands of application flows.

RINA research is experiment-intensive and typically makes use of GENI or FIRE [14]. The generation of large experiment topologies and their configuration is very time consuming, so it is important to have tools to automate these processes [15] [16]. Our experience with manual configuration and setup of even moderately sized RINA networks showed it is an error-prone, hard to reproduce process, that can also introduce unwanted dependencies on on-premise hardware.

In this work we present Rumba, an experimentation framework that builds on existing testbed frameworks (jFed [9] and emulab [17]) and extends them to provide a simple yet powerful API for experimenters to configure all the aspects of a RINA network, deploy it and run it on testbeds provided by GENI and FIRE.

Rumba is a Python library that allows the user to programmatically define (i) the physical connectivity graph of the network; (ii) how RINA layers are laid out on the nodes, without any restriction on layer membership and stacking geometry; (iii) the policies to be used by each layer; (iv) where and when distributed applications should run. Rumba already supports several testbeds and RINA implementations and is easily extendible.

In Section II Rumba requirements and objectives are detailed. Section III illustrates the Rumba software architecture. We evaluate the experiment setup times in Section IV. A demonstration scenario is presented in Section V. Finally, Section VI reports our conclusions.

II. Requirements

RINA networks can be seen as distributed applications that are layered on top of each other. A simple example of how RINA and its layering works is shown in Fig. 1. In this example, a client application β (on node B) and a client application δ (on node D) want to communicate with a server application α (on node A). The purpose of a layer in RINA is to provide an Inter-Process Communication (IPC) mechanism allowing other processes to allocate a *flow* over it. Internally, a layer consists of co-operating processes, called IPC Processes (IPCPs), depicted as grey circles in the Figure. *Enrolling an IPCP into a layer* is the process of authenticating with an existing member of the layer, obtaining the policies of the layer, and starting to function as a full-fledged member of the layer, i.e. being able to provide IPC resources to its users. The main difference between the layers is the *scope* over which they provide IPC. At the lowest layer, they simply provide an abstraction for the physical medium that is used, so here the scope is link-local (layers 1, 2 and 4 in the Figure). In the example, if the server α is registered in layer 4, then client β can use the services offered by layer 4 to allocate a flow to the server α. IPC Processes themselves can also use the services offered by the lower layers. The higher layers use the IPC services offered by the lower layers to provide IPC over a greater scope. So the same layer is repeated as many times as needed to achieve the required scope. In our example, if the server α is registered in layer 5, then client δ can use the services offered by layer 5 to communicate with the server α. A layer can be configured differently depending on the environment it is operating in.

In a nutshell, the objective of Rumba is to minimize the time spent on configuring advanced RINA networks and running large-scale experiments, and allow the user to get reproducible results.

Firstly, it should *provide an easy-to-use API*, so users can define all the aspects of their experiments with a single script. The obvious choice is to use a scriptable programming language like Python.

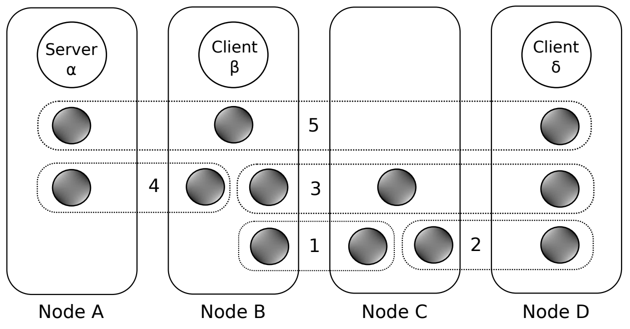

Fig. 1. A simple example of RINA layering. Scope for layers 1, 2 and 4 is local to a physical link. Scope for layers 3 and 5 is larger and spans multiple nodes.

Secondly, it should support arbitrary experiment network graphs, of any size. Using the API, the user can define how layers are stacked on top of each other, with no restrictions.

Thirdly, it should *support as many testbeds as possible*, interfacing with the most commonly used testbed management services like emulab and jFed. Such services allow instantiating a number of nodes (physical machines, virtual machines or containers) and connect them to form an arbitrary experiment infrastructure. The scriptable API offered by Rumba must be independent of any specific testbed managed by this service.

Fourthly, it should *support different prototypes*. The user must be able to easily select the prototype, while the framework takes care of the installation and configuration on the nodes. The API must be independent of the prototypes.

Fifthly, it should *provide simple interfaces to define the experiment execution*, i.e. (i) which nodes can run a client or server program; (ii) through which layers servers are reachable and (iii) when clients should run and how long they should run before terminating. Client scheduling time and duration can be defined either with a timestamp or in terms of statistical distributions, to emulate real world workloads.

Sixthly, once clients and server are done and the experiment is complete, it should *offer APIs to easily collect results*.

Finally, Rumba should *guarantee that experiments are as reproducible as possible*. Since an experiment is completely defined by a script, running the same script multiple times should lead to comparable results. Note that the testbed can select different physical nodes and resources (or concurrently host other experiments), so the actual results (such as throughput and latency figures) may differ across different runs of the same experiment. It is up to the experimenter to judiciously select the testbed, and interpret the results based on the full set of resources that the testbed can offer.

III. Design and implementation

To meet the requirements defined in Section II, Rumba is written in Python and built on a modular design, consisting of a core module with several plugin modules as shown in Figure 2. The core module contains functions operating on the abstract structure of the experiment, such as the nodes, their connectivity graph and the layers. The core only contains

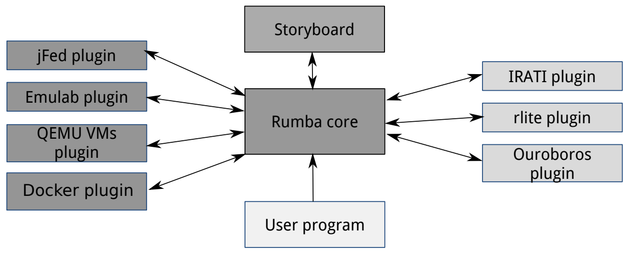

Fig. 2. High level architecture of Rumba

a general abstraction of a testbed and a prototype and any interaction with the actual testbed and prototype is handled via plugin modules. This design based on plugins also allows the Rumba library to be easily extended with support for additional testbeds and future prototypes. As an indication, the current plugins are between 200 and 450 lines of code.

A. Experiment model

Users interact with Rumba mainly through an `Experiment` object, which stores the network graph, the structure and configuration of all the layers, and references to the prototype and the testbed to be used. In more detail, the user specifies the testbed to be used through an instance of the `Testbed` class. The network structure is defined by a collection of instances of the `Node` and `DIF` classes (Distributed IPC Facility – DIF in short – is RINA terminology for a layer). For every node, the user has to pass the layers that they are a part of, and how the layers are stacked in the node, i.e. which layer registers in which layer. The user does not need to specify the IPCPs or their configuration, as this detail is figured out by Rumba. It is also possible to request a certain machine type for a node, i.e. a physical or virtual machine or container. Any specific testbed details are passed by the user through the `Testbed` class. Then, an instance of the `Experiment` object has to be created which links together the testbed, the layers and the nodes. On `Experiment` creation, Rumba does not instantiate any node nor tries to access the testbed. Instead it runs some graph algorithms to generate information that will be necessary during later stages.

First, it computes a topological ordering for the layers in the network, using Kahn's algorithm [18] on the *registrations graph*. The registrations graph is a directed acyclic graph (DAG) that contains a vertex for each layer and an edge from a layer x to layer y if and only if an IPC Process (IPCP) in layer y registers itself into the layer x. This graph represents the precedence relationship between layers: in particular, enrollments within a preceding layer must complete before enrollments within the succeeding layer can start. Figure 3 shows the graph for the layers in Figure 1. A cycle can occur because of user misconfiguration (infinite recursion); in this case the issue is detected and reported.

Second, Rumba derives the IPCPs that must be instantiated on each node, generating for each one the required configuration and commands to register to lower layers. Clearly, for each layer x, one (and only one) IPCP is generated for each node that is part of x. In Fig. 1 each circle represents an IPCP.

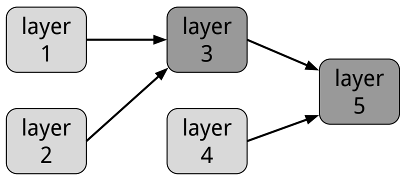

Fig. 3. Registrations graph for the layering of Fig 1. A possible topological order for the layers is $1, 2, 3, 4, 5$.

Finally, the sequence of enrollments is generated respecting the layer topological ordering, and in a way to avoid temporary network partitions in any layer. Within each layer, an arbitrary IPCP is elected to be the *bootstrapper* for that layer, i.e. it is the first node in the layer without the need to enroll, and it is configured as specified by the experimenter. All the other IPCPs in the layer enroll against the bootstrapper or a previously enrolled IPCP. In particular, the enrollment sequence within a layer is obtained through a simple Breadth First Search on the layer graph, starting from the bootstrapper; this strategy allows to avoid temporary network partitions. The global enrollment sequence is finally obtained by concatenating the list of all layers according to their topological ordering.

Once an `Experiment` object is initialized, the user can perform the following operations with subsequent API calls:

- *swap-in*: The testbed plugin is instructed to instantiate all the nodes and the physical (or virtual) links connecting them.
- *install-prototype*: The selected prototype is built and installed from sources on all the nodes, using the services offered by a prototype plugin. Every interaction with the nodes happens through SSH. Installation happens in parallel to speed up the process.
- *bootstrap-prototype*: The prototype plugin accesses each node to start and configure the RINA software to realize the RINA network specified by the `Experiment` object. In this phase all the IPCPs are created and configured, and enrollments are carried out in the order set during the previous stages.
- *swap-out*: The testbed plugin is asked to terminate all the nodes and links, and release all the testbed resources associated to the experiment.

A user script will normal carry out swap-in, install-prototype and bootstrap-prototype and then run the applications using the Storyboard services described in Section III-D. On script termination, the swap-out is invoked to release the testbed resources.

B. Testbed plugins

A testbed plugin wraps all the operations needed to acquire and release the compute, storage and network resources needed for an experiment. It provides an implementation for the swap-in and swap-out operations as described in Section III-A. The backend testbed management software provides operations to instantiate physical or virtualized nodes and setup Ethernet connectivity between pairs of nodes (e.g. using dedicated switches, VLANs or software L2 bridges).

Rumba currently supports any testbed accessible through the jFed and emulab interfaces; these testbeds are *remote*, since the nodes are instantiated in a remote cloud environment that is separate from the machine where the Rumba script runs. In addition to those, Rumba also supports two *local* testbeds. For the first local testbed each node is backed by a QEMU [19] virtual machine running locally (i.e. on the same host where the script runs). Nodes are interconnected using TAP devices (bridged together through a software L2 switch) or UDP sockets, as provided by QEMU, to emulate the physical network graph. The other local testbed instantiates a docker container for every node in the experiment. It is also interconnected using TAP devices and L2 software switches. The local testbeds should not be used to measure performance (since the resources of a single machine are too limited), but it is extremely useful to speed up developing and debugging Rumba scripts, due to its fast swap-in times in terms of seconds per node.

C. Prototype plugins

A prototype plugin wraps the operations required on each node to build a RINA prototype from sources, and setup and configure all the layers specified by the experiment. These operations are grouped together to implement the install-prototype and bootstrap-prototype methods described in Section III-A. Even in the bootstrap-prototype phase, the task of the prototype plugin is relatively simple, as it only needs to translate the list of abstract instructions (generated during the creation of an `Experiment` object) to prototype specific instructions. Such instructions lists are already sorted in the correct order (induced by the network structure), so the plugin does not need to worry about dependencies. This is also shown by the small code size of the prototype plugins, only a mere hundred lines of code specific to the prototype is required.

In more detail, a prototype plugin needs to process three lists of instructions: (i) IPCP creation and configuration; (ii) registration of IPCPs in their lower layers; and (iii) enrollment of IPCPs to layers they belong to.

Currently, Rumba supports three implementations (all of them free and open source):

1) IRATI [11], the first complete RINA implementation developed by the EU-funded RINA projects, targeted at GNU/Linux systems.
2) rlite [20], a lightweight RINA implementation for GNU/Linux operating systems.
3) Ouroboros [21], a user-space recursive network implementation with a focus on portability. It is written in C89 and works on any POSIX.1-2001 compliant system.

D. Storyboard

After the bootstrap-prototype phase, the recursive network is ready to run distributed applications. This task is achieved by means of the Rumba Storyboard module. The Storyboard is able to emulate realistic network traffic by starting and stopping server and client programs at specific nodes across the network. The user specifies: the duration of the Storyboard, the server programs and their command line arguments, through which layers a particular server is reachable, the node where servers run, the client programs with their corresponding command line arguments, and the nodes where clients run. Programs can be terminated by a custom command (if specified), or by the SIGTERM signal. As an example, a server program may be an Apache server, and the client programs could be the Firefox and Chrome web browsers. The emulation can be either generated as a random process (according to given statistical distributions), replayed from a previously generated Storyboard, or the schedule can be manually specified by the user. In addition to starting and stopping user applications, the Storyboard is also able to emulate network failure and recovery events, such as link up/down events or node up/down events. This is very useful for experiments involving load balancing, resiliency, consistency and high availability.

E. Other implementation details

Rumba has minimal dependencies. On any machine with Python version greater than 3.2 installed, the only extra dependency is Paramiko, which is the SSH library used by Rumba to access the nodes in the experiment. In case the installed Python version is smaller than 3.2, some wrapper packages are required.

Rumba also provides the `rumba-access` script to access the nodes after swap-in, which wraps any testbed specific details for accessing the nodes, such as proxy commands. The user can simply invoke `rumba-access` followed by the node name as defined in the user's experiment script to access the node.

IV. Operation

Rumba has built-in support for performance tests, with automated client/server log recovery and swap-in, install, bootstrap and swap-out timers. This shortens the time needed to retrieve results and allows the user to focus on results analysis. We measured the timings of swap-in, installing a prototype, bootstrapping it and swapping the experiment back out for the example in Figure 1; we repeated the measurement ten times. We did not include the time it takes for a Storyboard to complete, since this is configurable by the experimenter. Bringing up this scenario manually could easily take up to a day of work, as the user has to: use a GUI to create the network physical connectivity graph, instantiate the resources on the testbed, login to the nodes one by one and install the prototype, manually figure out the registrations and enrollment order of IPCPs, execute the commands one by one, and check the logs for any prototype errors due to misconfiguration. The results of these measurements with the rlite prototype for different testbeds are shown in Table I, together with their 95% confidence intervals. We did not perform tests for the docker testbed, since rlite does not yet support namespaces in the kernel (a required feature for containers to work).

TABLE I
TIMINGS FOR A RUMBA EXPERIMENT (IN SECONDS)

	Swap-in	**Install**	**Bootstrap**	**Swap-out**
QEMU	20.92 ± 0.03	N/A	22.36 ± 0.04	0.86 ± 0.03
Emulab (Virtual Wall)	94.27 ± 4.33	132.49 ± 2.33	25.31 ± 0.09	1.01 ± 0.04
jFed (Virtual Wall)	186.97 ± 9.02	128.39 ± 1.09	24.57 ± 0.08	31.63 ± 8.93
jFed (Exogeni)	141.88 ± 26.37	156.31 ± 36.77	42.73 ± 1.09	27.17 ± 0.99

In the case of the QEMU testbed, prebuilt VM images are used of around 35 MB in size. These images contain the latest versions of each prototype pre-installed and are downloaded automatically to a cache directory if they are not present on the system running the experiment. Since the images come with pre-installed prototypes, there is no measurement for the installation phase. The total time of the experiment on the QEMU testbed is 44.15 seconds, so this is an ideal setup for debugging and tweaking experiments or running preliminary tests before deploying it on an real testbed (thus conserving valuable testbed resources). The confidence intervals are also very low since the experiment executes locally, which minimizes uncontrollable variables. The QEMU testbed can be leveraged for catching any deficiencies in a test setup early-on. It is also ideal for integration testing of the prototypes. The main limitation of this testbed is that it is limited by the computer it is running on; every new node requires a new VM, which consumes CPU power and a significant amount of RAM. Furthermore, any performance results will be impacted by the host Operating System (schedulers, disk I/O, memory architecture, etc). For such measurements, we need to rely on large testbeds.

The imec iLab.t Virtual Wall [22] consists of a few hundred physical machines and supports both the emulab interface and the jFed interfaces. We ran tests using both interfaces on the Virtual Wall 2 authority. As can be seen, the timings of install and bootstrap are very similar, which is to be expected, since the same hardware is used in both tests. The main difference in these tests are in swap-in and swap-out time. In the case of emulab the total time spent provisioning resources is 94.27 seconds, whereas in the case of jFed this is 186.97 seconds, almost double the amount of time. We believe this difference in swap-in time between emulab and jFed is because via emulab a direct interface is provided to the Virtual Wall, whereas via jFed some other steps are required specific to jFed (information has to be shared with other federated testbeds). The swap-out time for emulab is also a lot lower, however, this is due to the fact that emulab does not wait until all resources are released. In case an emulab interface is provided by the testbed, we suggest using that one to minimize the experiment time, since only around 4 minutes are needed per experiment on getting the fully functioning RINA network depicted in Figure 1 on physical machines.

The final test we performed was with jFed on the ExoGENI testbed [23], which itself is a federation of independent cloud sites. The testbed consists of several Virtual Machines in different locations around the world, but mainly concentrated in the US. As can be seen, the swap-in time is on average 141.88 seconds, which is comparable to the swap-in time of jFed with the Virtual Wall, yet the 95% confidence interval for this result is very high, 26.37 seconds. We believe this is due to the fact that ExoGENI is very distributed, and different sites have to be contacted to provision the required resources. If the availability of resources on a single location is low, then another location has to be contacted, and so on until the required amount of nodes is found. The install and bootstrap times are similar to the ones on the Virtual Wall. They are just a bit slower since the experiment is running on Virtual Machines. The high confidence interval for installing the prototype can be explained by the fact that Rumba installs some required packages via aptitude in the case of an Ubuntu or Debian image; however, the Ubuntu image used by ExoGENI sometimes is busy with automatic software updates, so Rumba needs to wait until the aptitude lock is released before installing its own packages. The total time an experiment on ExoGENI takes is 368.10 seconds. However, the main advantage of the ExoGENI testbed is its abundance of resources, since most of the time it has 800 VMs or more available.

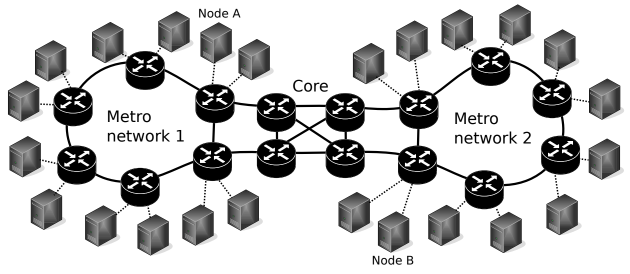

Fig. 4. Physical connectivity graph of the converged network operator demo.

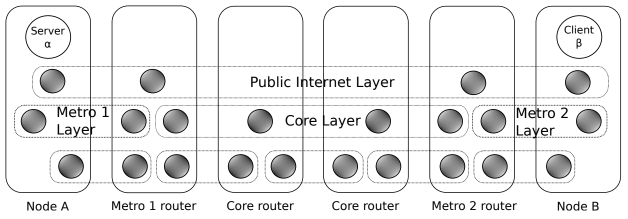

Fig. 5. Recursive layering of the converged network operator demo

V. DEMONSTRATION

We now describe a larger scale experiment with Rumba than the one depicted in Figure 1, namely a converged network operator's network. Figure 4 shows the physical connectivity graph of the operator's network. It consists of servers and clients connected to a metro network, which in turn is connected through a core network to other metro networks with more servers and clients connected to them. This network is easily generalized, as it can be enlarged at will by adding more metro networks and by enlarging the core network. We want to demonstrate this scenario and show that Rumba dramatically cuts back on the setup time.

How such a network can be modelled with RINA is depicted in Figure 5 as a 2D projection for the case of a client β that wants to communicate with a server α in a different metro network. Note that we purposely kept the design relatively simple; a real design might be more complicated. At the bottom level, each physical link is abstracted away as a layer over Ethernet. The routers in the metro network and the servers and clients will all have an IPCP in the metro layer. In case a client wants to talk to a server that is located in the same metro network, it can simply use the metro layer to allocate a flow over. The core routers and the metro border routers will have an IPCP in the core layer. This is done so that the management of the core layer is self-contained. Each layer is its own management domain.

In case a server and client in different metro networks want to communicate with each other, they need another layer with even greater scope, here conveniently called the public internet layer. All servers, clients, and metro border routers will have an IPCP in the public internet layer. When the client β allocates a flow to server α, it will do so over the public internet layer. A flow allocation request will be sent in the public internet layer by node B's IPCP. The next hop, the metro 2 border router's IPCP, is reached by using the IPC services provided by the metro 2 layer. The metro 2 border router's IPCP will then use the services offered by the core layer to reach the next hop, i.e. the metro 1 border router's IPCP. Finally, the metro 1 border router's IPCP will forward the request to node A's IPCP. Once a positive reply is sent back and acknowledged, the flow is allocated, so that β and α can start to communicate.

We will demonstrate this scenario by installing and bootstrapping the different prototypes on different testbeds. Every node connected to a metro network will be available as both a server and a client. A Storyboard will thus be run to emulate real network traffic over the converged network operator network.

VI. Conclusion

In this paper we presented Rumba, a free open source experimentation framework for installing and bootstrapping recursive networks on various testbeds. We explained the layering structure of RINA and outlined the requirements for a RINA experimentation framework, upon which we based the design of Rumba. We described Rumba's modular design which abstracts away any prototype or testbed internals, enabling easy integration of several prototypes and testbeds. We showed that by using Rumba, the experiment execution time is drastically reduced. We presented a more elaborate demonstration of RINA, namely how it can be deployed on a converged network operator network. The repository holding Rumba can be found at [24], or it can be installed through pip, the Python package manager by issuing `pip install rumba`. In the future, we intend to extend Rumba with more testbeds; for instance, work is being done to add support for OpenStack. We also intend to implement traffic limitation and degradation, to emulate subpar links with packet loss, delay, low bandwidth and high bit error rate.

Acknowledgment

This work was partially funded by the European Commission through the FIRE+ H2020 Programme, grant no 687871, ARCFIRE.

References

[1] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[2] (2017, Dec.) Open Networking Foundation. [Online]. Available: https://www.opennetworking.org/platforms

[3] (2017, Dec.) OpenDaylight. [Online]. Available: https://www.opendaylight.org

[4] L. Chiaraviglio, L. Amorosi, S. Cartolano, N. Blefari-Melazzi, P. Dellolmo, M. Shojafar, and S. Salsano, "Optimal superfluid management of 5G networks," in *2017 IEEE Conference on Network Softwarization (NetSoft)*, July 2017, pp. 1–9.

[5] S. Dräxler, H. Karl, M. Peuster, H. R. Kouchaksaraei, M. Bredel, J. Lessmann, T. Soenen, W. Tavernier, S. Mendel-Brin, and G. Xilouris, "SONATA: Service programming and orchestration for virtualized software networks," in *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*, May 2017, pp. 973–978.

[6] (2017, Dec.) The GENI website. [Online]. Available: http://www.geni.net/

[7] (2017, Dec.) The FIRE website. [Online]. Available: http://www.ict-fire.eu

[8] W. Vandenberghe, B. Vermeulen, P. Demeester, A. Willner, S. Papavassiliou, A. Gavras, M. Sioutis, A. Quereilhac, Y. Al-Hazmi, F. Lobillo, F. Schreiner, C. Velayos, A. Vico-Oton, G. Androulidakis, C. Papagianni, O. Ntofon, and M. Boniface, "Architecture for the heterogeneous federation of future internet experimentation facilities," in *2013 Future Network Mobile Summit*, July 2013, pp. 1–11.

[9] (2017, Dec.) The jFed website. [Online]. Available: https://jfed.ilabt.imec.be/

[10] 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*. ACM, 2008, p. 67.

[11] 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*, vol. 28, no. 2, pp. 20–25, 2014.

[12] V. Maffione, F. Salvestrini, E. Grasa, L. Bergesio, and M. Tarzan, "A Software Development Kit to exploit RINA programmability," in *Communications (ICC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 1–7.

[13] (2017, Dec.) The ARCFIRE website. [Online]. Available: http://ict-arcfire.eu/

[14] Y. Wang, F. Esposito, and I. Matta, "Demonstrating RINA using the GENI Testbed," in *Research and Educational Experiment Workshop (GREE), 2013 Second GENI*. IEEE, 2013, pp. 93–96.

[15] Z. Fei, Q. Xu, and H. Lu, "Generating large network topologies for GENI experiments," in *IEEE SOUTHEASTCON 2014*, March 2014, pp. 1–7.

[16] D. Duplyakin and R. Ricci, "Introducing configuration management capabilities into CloudLab experiments," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, April 2016, pp. 39–44.

[17] (2017, Dec.) The Emulab website. [Online]. Available: http://www.emulab.net/

[18] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[19] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[20] (2017, Dec.) The rlite repository. [Online]. Available: https://github.com/vmaffione/rlite/

[21] (2017, Dec.) The Ouroboros website. [Online]. Available: https://ouroboros.ilabt.imec.be/

[22] (2017, Dec.) imec iLab.t Virtual Walls. [Online]. Available: http://doc.ilabt.iminds.be/ilabt-documentation/virtualwallfacility.html

[23] (2017, Dec.) The ExoGENI website. [Online]. Available: http://www.exogeni.net/

[24] (2017, Dec.) The Rumba repository. [Online]. Available: https://gitlab.com/arcfire/rumba