Programming Routing Policies for Video Traffic

Yuefeng Wang Nabeel Akhtar Ibrahim Matta
Computer Science Department, Boston University
Boston, MA 02215
{wyf, nabeel, matta}@bu.edu

***Abstract*—Making the network programmable simplifies network management and enables network innovations. The Recursive InterNetwork Architecture (RINA) is our solution to enable network programmability. ProtoRINA is a user-space prototype of RINA and provides users with a framework with common mechanisms so a user can program recursive-networking policies without implementing mechanisms from scratch.**

In this paper, we focus on how routing policies, which is an important aspect of network management, can be programmed using ProtoRINA, and demonstrate how ProtoRINA can be used to achieve better performance for a video streaming application by instantiating different routing policies over the GENI (Global Environment for Network Innovations) testbed, which provides a large-scale experimental facility for networking research.

I. Introduction

Computer networks are becoming more and more complex and difficult to manage, and the research community has been expending a lot of efforts to address the complexity of network management. Making the network programmable, as exemplified by recent research in Software Defined Networking (SDN) [1], simplifies network management and enables network innovations. The Recursive InterNetwork Architecture (RINA) [2], [3] is our solution to enable network programmability, and as a consequence, achieve better network manageability.

The RINA architecture is a new architecture that inherently solves shortcomings of the current (TCP/IP) Internet architecture—for example, TCP/IP's path-dependent addressing makes it hard to support mobility, and its rudimentary "best-effort" service does not accommodate QoS requirements. By separating mechanisms and policies, RINA enables policy-based programming of the network. Also RINA's recursion of the inter-process communication (IPC) service over different scopes yields a better service management framework [4]. ProtoRINA [5], [6], [7] is Boston University's user-space prototype of RINA. ProtoRINA offers a framework with common mechanisms that enables users to program recursive-networking policies without implementing mechanisms from scratch.

GENI (Global Environment for Network Innovations) [8] is a nationwide suite of infrastructure that enables research in networking and distributed systems, and it supports experimentation with future network architectures, such as RINA, for possible deployment.

In this paper we explain how routing policies, which is an important aspect of network management, can be programmed using ProtoRINA. Through experiments over GENI, we demonstrate how ProtoRINA can be used to achieve better performance for a video streaming application with appropriate routing policies. The rest of the paper is organized as follows. Our RINA architecture and ProtoRINA are briefly explained in Section II. Section III describes how routing policies can be programmed using ProtoRINA. Experiments over the GENI testbed are presented in Section IV. Section V describes our experiences and lessons learned from using GENI. Section VI concludes the paper with future work.

II. Recursive InterNetwork Architecture

A. Recursive Architecture

The Recursive InterNetwork Architecture (RINA) [2], [3] is a new network architecture based on the fundamental principle that *networking is Inter-Process Communication (IPC) and only IPC*. It is designed based on two main principles: (1) divide and conquer (recursion), and (2) separation of mechanisms and policies.

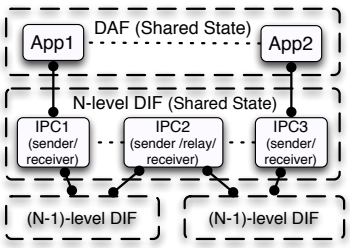

Fig. 1. RINA overview

As shown in Figure 1, a set of distributed application processes that cooperate to perform a certain function, such as communication service or weather forecast, is called a *Distributed Application Facility (DAF)*. A *Distributed IPC Facility (DIF)* is a specialization of a DAF that provides communication service. A DIF is a collection of distributed IPC processes with shared states, and it provides communication service over a certain scope ($i.e.$, range of operation). Recursively, a higher-level DIF providing communication service over a larger scope is formed based on lower-level DIFs that provide smaller-scope communication services.

For two application processes to communicate, they have to connect through a common underlying DIF of IPC processes. If there is no such common DIF, then one must be dynamically formed. In the latter case, a directory service resolves the destination application name to an IPC process inside a DIF through which the destination application can be reached. The source IPC process then joins the DIF through that IPC member following an enrollment procedure [9]. Once the source IPC process is enrolled, it is assigned an internal

private address and initialized with all configurations that are needed for operations inside the DIF, culminating in a transport flow between the higher-level application processes. Addresses inside a DIF are not exposed to the application processes, and there are no well-known communication ports.

RINA separates mechanisms and policies, and IPC processes use the same mechanisms but may use different policies in different DIFs. For example, different DIFs can have their own routing policies or authentication policies. RINA simplifies the network protocol stack by using only two protocols: the Error and Flow control Protocol (EFCP) is used for data transfer, and the Common Distributed Application Protocol (CDAP) is used by management or user applications.

B. ProtoRINA

ProtoRINA [5], [6], [7] is Boston University's user-space prototype of RINA. ProtoRINA enables the programming of recursive-networking policies. It can be used by researchers as an experimental tool to develop (non-IP) user and management applications, and also by educators as a teaching tool for networking and distributed systems courses. The current version implements the basic recursive structure and management architecture of RINA, but it is not a complete implementation, specifically the EFCP protocol is not fully implemented and we keep modifying and adding new elements.

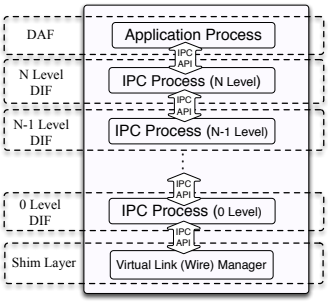

Fig. 2. RINA node

1) RINA Node: As shown in Figure 2, a *RINA node* is a host (or machine) where application processes and IPC processes reside. Application processes or high-level IPC processes communicate with their peers using the communication service provided by low-level IPC processes that act as points of attachment. In the current implementation, physical connectivities between IPC processes in level-0 DIFs are emulated by TCP connections via a shim layer.

Each RINA node has a configuration file that includes the information of all processes residing on it, and each process also has its own configuration file. When a RINA node is started, all processes on the node are bootstrapped based on their own configuration files.

2) IPC Process: The components of an IPC process and how they interact with each other through RINA APIs are shown in Figure 3. Each IPC process has a *Resource Information Base (RIB)* that stores its view of all information related to the operations inside the DIF. The *RIB Daemon* helps other components access information stored in the local RIB through the *RIB API* or in a remote IPC's RIB through CDAP messages. The *IPC Resource Manager (IRM)* manages the use of underlying low-level DIFs to allocate and maintain the connections between the IPC process and its peers.

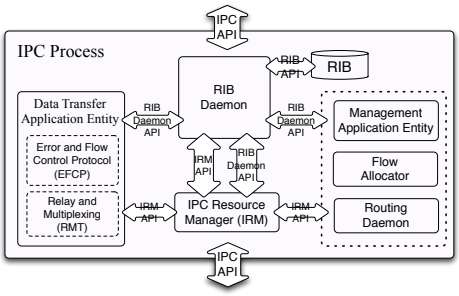

Fig. 3. RINA IPC process: components and APIs

3) RINA APIs: The *RIB Daemon API* and *IRM API* (shown in Table I) are provided to users to write their own applications or define new network management policies.

RIB Daemon API
public int createEvent(SubscriptionEvent subscriptionEvent);
public void deleteEvent(int subscriptionID);
public Object readSub(int subID);
public void writePub(int pubID, byte[] obj);
IRM API
public int allocateFlow(Flow flow)
public void deallocateFlow(int handleID);
public void send(int handleID, byte[] msg) throws Exception;
public byte[] receive(int handleID);

TABLE I
RIB Daemon API and IRM API provided by ProtoRINA

The RIB Daemon API is based on a publish/subscribe model, but it also supports the traditional pulling mechanism to retrieve information. It supports the creation and deletion of a subscription event (a *Pub* or *Sub* event), the retrieval of information through a *Sub* event, and the publication of information through a *Pub* event. An *event ID* is returned when a new event is created. Attributes used in the *subscription event object* include: event type (Pub or Sub), subscription ID, subscriber member list, update period, attributes list and attributes values. The IRM API allows allocating/deallocating a connection (flow), and sending/receiving messages over existing connections. Once a new connection is created, a *handle ID* is returned, which can be used to send and receive messages. The *flow object* contains the attributes of a connection, and these attributes include: source/destination application name, source/destination address and connection end-point ID, and quality-of-service.

More details about the IPC process, its components and RINA APIs, can be found in [6].

III. Programming Routing Policies

In this section, we describe how routing policies can be programmed using ProtoRINA and how policy-based routing can be achieved, where routing configurations can be easily expressed in terms of high-level policies.

A. Routing Mechanisms

The *Routing Daemon* (shown in Figure 3) is the component that is responsible for routing inside the DIF. The routing

policies of an IPC process can be specified in its configuration file, or be initialized by existing members inside the DIF after the IPC process is enrolled into the DIF. Particularly, IPC processes use the same mechanisms provided by the Routing Daemon, but they may have different routing policies in different DIFs. All IPC processes inside the same DIF have to follow the same routing policies (for example, either link-state or distance-vector routing) in order to collaborate and perform the routing functionality for the DIF.

In ProtoRINA routing policies can be easily configured using the configuration file. As an example, in Table II we show the routing policies specified in the configuration file of an IPC process. In this example, the IPC process uses distance-vector routing. Distance-vector information is collected from its neighbors every 5 seconds. Neighbor connectivities are checked every two seconds. The path cost is calculated using hop count.

```
rina.routing.protocol = distanceVector
rina.distanceVectorUpdatePeriod = 5
rina.checkNeighborPeriod = 2
rina.linkCost.policy = hop
```

TABLE II
Routing policies specified in the IPC process' configuration file

The Routing Daemon collects routing information (including neighbor connectivity, link cost, link state or distance vector) through Pub/Sub events using the RIB Daemon API (shown in Table I) provided by the RIB Daemon.

```
SubscriptionEvent event1 = new SubscriptionEvent(
 EventType.SUB, period1, ``checkNeighborAlive", neighbor);
int eventID1 = ribDaemon.createEvent(event1);

SubscriptionEvent event2 = new SubscriptionEvent(
 EventType.SUB, period2, ``distanceVector", neighbor);
int eventID2 = ribDaemon.createEvent(event2);
```

TABLE III
An example that shows creation of two *sub* events using RIB Daemon API

As an example, we show how distance-vector routing is supported through Pub/Sub events. For a DIF that uses distance-vector routing, after an IPC process joins the DIF, for all its direct neighbors, the Routing Daemon creates two *Sub* events shown in Table III: (1) the *checkNeighborAlive* event allows the Routing Daemon to collect direct neighbors' connectivity status (up/down) and also measure link cost; and (2) the *distanceVector* event provides the Routing Daemon with the distance vector from each of its neighbors. Through these two events, the Routing Daemon collects the distance vector information, and then builds the forwarding table using the Bellman−Ford algorithm (along with split horizon and poison reverse techniques to prevent routing loops and the counting-to-infinity problem).

B. Routing Policies

1) Routing Computation Policy: Two classical routing computation policies are supported: link-state and distance-vector. Also ProtoRINA allows users to set different update frequencies (as shown in the example in Table II) for different Pub/Sub events that are used to collect routing information. For a more static network, routing information can be collected less frequently, while for a more dynamic network, a higher frequency is needed.

2) Link Cost Policy: ProtoRINA currently supports two link-cost policies. Path cost can be calculated using hop cost, or using jitter (packet delay variation). In our implementation, delay jitter of each link is computed as described in [10], and the path cost is the sum of jitter on each link along that path.

3) DIF Formation Policy: ProtoRINA also allows users to define the network topology. For the same physical topology, users can either build a single-level DIF or multi-level DIFs, and the number of IPC processes in each DIF can also be configured. For example, a two-level DIF topology can apply to a situation where each level-0 DIF (lower-level over smaller scope) is a local network (where jitter may be small enough to be ignored), but a level-1 DIF (higher-level over larger scope) spans a wide-area network (where jitter on links connecting the local networks may be more significant and affect application performance).

In our previous work [11], we show that DIF formation can be configured to achieve less routing overhead and faster convergence time in the face of link failures. In this paper, we configure a two-level DIF topology and set the link-cost policy to "jitter" so as to route video traffic over the least-jitter path and improve video performance.

C. Programming New Policies

ProtoRINA provides routing policy holders for users to define their own routing policies using the RIB Daemon API (Table I), which helps collect routing information inside the DIF. In the current version, we support two routing policy holders. The *routing computation policy holder* allows the creation of new routing computation policies (other than link-state or distance-vector, *e.g.* path-vector), and the *link cost policy holder* allows the creation of new link-cost policies (other than hop or jitter, *e.g.* delay). Once new routing policies are added, users can simply apply new policies by specifying them in the configuration file.

What's more, using the RINA APIs (RIB Daemon API and IRM API), users can easily add other RINA policies (such as authentication policy, addressing policy, and resource allocation policy), and develop their own user or management applications without implementing mechanisms from scratch.

IV. Video Streaming over ProtoRINA

In this section, we present our experiments on the GENI testbed. As an example of application-specific policies, we demonstrate how an appropriate link-cost policy can be used to achieve better performance for video streaming applications.

A. GENI Testbed

GENI (Global Environment for Network Innovations) [8] is a nationwide suite of infrastructure that enables research and education in networking and distributed systems. GENI supports large-scale experimentation with advanced protocols for data-centers, clouds, mobile and SDN networks, *etc*. And it supports experimentation with future network architectures, such as RINA, for possible deployment. Users (researchers and

educators) can obtain computing resources, such as virtual machines (VMs) and raw PCs, from different physical locations (GENI aggregates) that are layer-2 connected, and they can connect these computing resources via layer-2 links (stitched VLANs) or layer-3 links (Generic Routing Encapsulation tunnels).

Users can reserve and manage GENI resources through the GENI Portal [12]. The GENI Portal allows users to login using their home institution's credentials, and after logging in, users can reserve GENI resources using tools such as Flack and Omni, and monitor resources using tools such as GENI Desktop and LabWiki. Figure 4 shows an example of a GENI topology displayed in Flack.

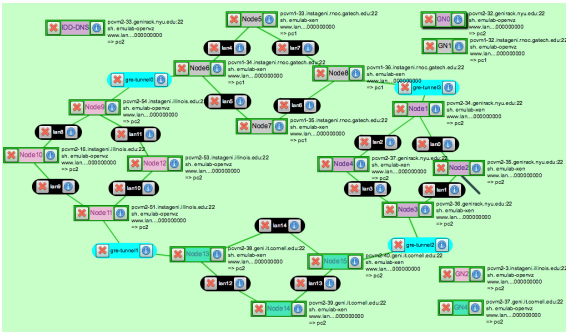

Fig. 4. GENI topology shown in Flack, a GUI tool of the GENI Portal

We have used a number of these GENI tools, including GENI Portal, Flack, Omni (Stitcher), and GENI Desktop. Details of our experiences with the GENI testbed are presented in Section V.

B. Video Streaming on Legacy Hosts

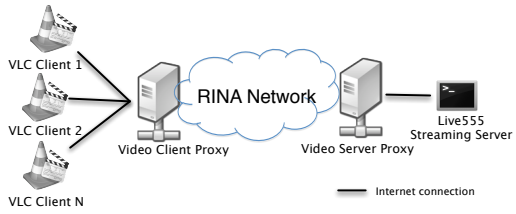

Fig. 5. Video clients (VLC players) are connected to the video streaming server (Live555 streaming server) through RINA proxies over a RINA network

In order to support a video streaming application running on legacy hosts, two video proxies are used as shown in Figure 5. The video client proxy is connected to the video clients over the Internet, and the video server proxy is connected to the video streaming server also over the Internet. These two proxies are connected over a RINA network that is composed of DIFs, where routing policies (and other policies) can be easily configured for each DIF. Video proxies support the Real-Time Streaming Protocol (RTSP) and redirect all traffic between the video clients and the video streaming server to the communication service provided by the RINA network. The details of these two proxies can be found in [13].

C. Experimental Design

As shown in Figure 6, we reserve GENI resources (VMs, VLANs, and GRE tunnels) from four GENI aggregates (Georgia Tech, UIUC, NYU, and Cornell University). VMs across aggregates are connected using GRE tunnels, and VMs in the same aggregate are connected using VLANs. Each RINA node (Section II-B1) is running on a GENI VM, and we use 13 RINA nodes (`Node 1` to `Node 13`). `Node 1`, `Node 2`, and `Node 3` are running on VMs from the NYU aggregate. `Node 4`, `Node 5`, `Node 6`, and `Node 7` are running on VMs from the Georgia Tech aggregate. `Node 8`, `Node 9`, and `Node 10` are running on VMs from the UIUC aggregate. `Node 11`, `Node 12`, and `Node 13` are running on VMs from the Cornell aggregate.

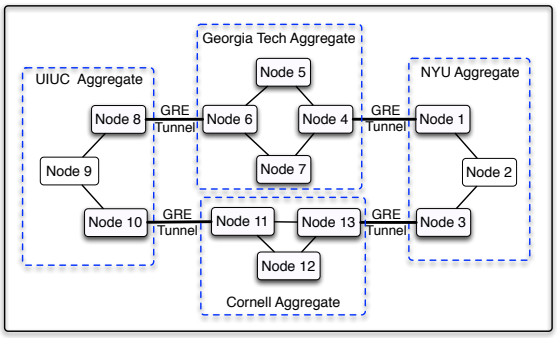

Fig. 6. Each RINA node is running on a GENI VM. RINA nodes across aggregates are connected via GRE tunnels, and RINA nodes in the same aggregate are connected via VLANs

1) DIF Formation Policy: To provide communication service for the video client proxy located at `Node 9` and the video server proxy located at `Node 2`, different DIF formation policies can be used to generate different DIF topologies. In our experiment, we only use the DIF formation policy shown in Figure 7, and focus on different link-cost policies.

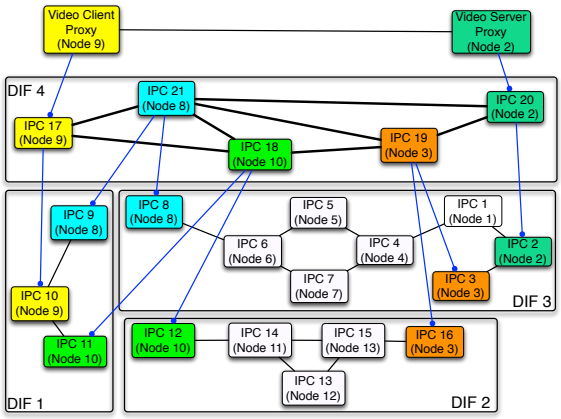

Fig. 7. Video client proxy (on `Node 9`) and video server proxy (on `Node 2`) communicate through a level-1 DIF (`DIF4`), which is built on top of three level-0 DIFs (`DIF1`, `DIF 2`, and `DIF 3`)

As shown in Figure 7, there are three level-0 DIFs (`DIF 1`, `DIF 2`, and `DIF 3`), and one level-1 DIF (`DIF 4`). `DIF 1` has three members: `IPC 9` (on `Node 8`), `IPC 10` (on `Node 9`), and `IPC 11` (on `Node 10`). `DIF 2` has five members: `IPC 12` (on `Node 10`), `IPC 13` (on `Node 12`) , `IPC 14` (on `Node 11`), `IPC 15` (on `Node 13`), and `IPC 16` (on `Node 3`). `DIF 3` has eight members: `IPC 1` (on `Node 1`),

IPC 2 (on Node 2) , IPC 3 (on Node 3), IPC 4 (on Node 4), IPC 5 (on Node 5) , IPC 6 (on Node 6), IPC 7 (on Node 7), and IPC 8 (on Node 8). DIF 4 has five members: IPC 17 (on Node 9), IPC 18 (on Node 10), IPC 19 (on Node 3) , IPC 20 (on Node 2), and IPC 21 (on Node 8).

Application processes (or IPC processes) talk to their peers using underlying IPC processes of lower-level DIFs on the same RINA node. In Figure 7, the video client proxy on Node 9 uses IPC 17, which recursively uses IPC 10. The video server proxy on Node 2 uses IPC 20, which recursively uses IPC 2. IPC 21 on Node 8 uses IPC 8 and IPC 9, IPC 18 on Node 10 uses IPC 11 and IPC 12, and IPC 19 on Node 3 uses IPC 3 and IPC 16.

So the video client proxy and video server proxy communicate through a connection supported by the underlying DIF 4, which recursively uses the communication services provided by three level-0 DIFs. The connection between the video server proxy (using IPC 20) and the video client proxy (using IPC 17) is mapped to a path inside DIF 4, and each link in DIF 4 is supported by a level-0 DIF.

2) Link Cost Policy: As mentioned earlier, RINA separates mechanisms and policies. Different DIFs (at different levels) use the same mechanisms but can use different policies. In our experiment, the three level-0 DIFs use hop as the link-cost policy. For the higher-level DIF, DIF 4, which directly provides communication service between the two video proxies, we test two link-cost policies: hop and jitter, and observe how they affect the performance of the video application.

3) Other Policies: All four DIFs use the link-state routing computation policy with the same frequency for collecting link-state information from neighbors (every two seconds).

Finally, we use the network emulation tool, *NetEm* [14], to emulate link delay and jitter. Delay (300ms) with variation (±200ms) is emulated on two physical links between RINA nodes (Node 1–Node 2, and Node 8–Node 9), and this emulation leads to jitter on link IPC 9–IPC 10 in DIF 1, and on link IPC 1–IPC 2 in DIF 3. This in turn is reflected as link jitter in the higher-level DIF 4, where four links (IPC 17–IPC 21, IPC 18–IPC 21, IPC 20–IPC 21, and IPC 19–IPC 21) exhibit jitter that they inherit from underlying paths.

D. Experimental Results

As we can see from Figure 7, the connection between the video server proxy and video client proxy can be routed via one of the seven possible loop-free paths inside DIF 4 between IPC 20 and IPC 17. These paths are: *path 1* (IPC 20–IPC 21–IPC 17), *path 2* (IPC 20–IPC 19–IPC 18–IPC 17), *path 3* (IPC 20–IPC 21–IPC 19–IPC 18–IPC 17), *path 4* (IPC 20–IPC 21–IPC 18–IPC 17), *path 5* (IPC 20–IPC 19–IPC 21–IPC 17), *path 6* (IPC 20–IPC 19–IPC 21–IPC 18–IPC 17), and *path 7* (IPC 20–IPC 19–IPC 18–IPC 21–IPC 17).

Figure 8 shows the path jitter of *path 1* (least-hop path) and *path 2* (least-jitter path). If jitter is used as link cost, the connection between IPC 20 and IPC 17 is routed on *path 2*. On the other hand, if hop is used as link cost, the connection is routed on *path 1*.

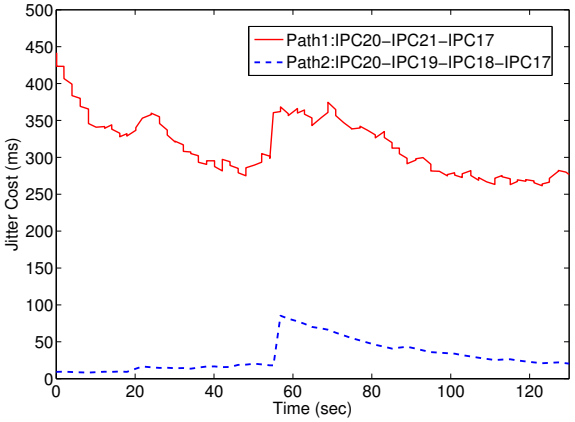

Fig. 8. Path jitter of *path 1* (the least-hop path) is larger than *path 2* (the least-jitter path)

For video applications, it is important to choose a path with least jitter, otherwise video quality degradation is observed. We indeed observe that the video freezes more often when using hop rather than jitter as the link-cost policy.

Figure 9 shows the instantaneous jitter from the video server proxy to the client proxy as the video server streams the same video to a player client. We can see that under least-jitter routing, the RTP packets experience much less jitter compared to least-hop routing[1].

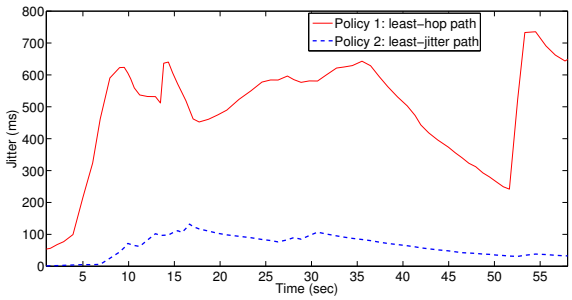

Fig. 9. Instantaneous jitter from the video server proxy to the client proxy when DIF 4 uses hop or jitter as link cost

V. GENI Experiences and Lessons Learned

We have been running experiments on GENI for the past two years, which has been instrumental to our efforts toward testing and improving our RINA architecture. GENI has evolved considerably over time, and new tools are frequently being added and updated. In this section, we present our experiences with GENI — these experiences are not limited to experiments in this paper.

A. Experiments Across Aggregates

Our experiments involve resources from a single aggregate and from multiple aggregates at different locations. There are

[1]To compute jitter, we sample RTP packets at the rate of one every 60 packets.

two ways to connect resources across different aggregates: GRE tunnels and stitched VLANs. GRE tunnels provide layer-3 connectivity using IP-in-IP encapsulation, while stitched VLANs provide layer-2 VLAN connectivity between different aggregates.

GRE tunneling is more easier to use as (1) it can be easily created through Flack, a GUI tool for resource reservation, and (2) it is more likely to succeed when reserving resources. On the other hand, stitched VLAN connectivity is a little bit complicated. It is not supported in Flack and we have to use Omni, a command-line tool, along with the *rspec* file (a configuration file describing the resource request). Also reserving stitched VLAN is more likely to fail especially when involving more than two aggregates, and there are fewer aggregates that support stitched VLAN. The recently released version of Omni (v2.6) has improved the success rate of establishing a stitched VLAN, however it would still be convenient if Flack can support stitched VLAN reservation.

GRE tunnels have some drawbacks. Since GRE tunnels use IP-in-IP encapsulation, they introduce extra overhead that may not be desirable for some experiments. Also for overlay experiments running over TCP, the extra header due to the encapsulation may cause IP fragmentation that leads to TCP performance degradation. In our experiments, we have manually configured the Maximum Transmission Unit (MTU) size from the default 1500 bytes to 1400 (or less) bytes for Network Interface Cards (NICs) on both ends of the GRE tunnel in order to avoid IP fragmentation and hence improve TCP performance.

When creating GRE tunnels and stitched VLANs, we can specify parameters such as capacity, packet loss, and latency, however these settings are not fully supported and we had to use emulation tools such as NetEm [14] in our experiments. It would be helpful if GENI can support the parametrization of GRE and VLAN connections.

B. Time Needed For Resource Reservation

As we run complicated experiments that involve more GENI resources, we found that it may take a lot of time (tens of minutes) to successfully reserve resources. Flack can become very slow when building complicated GENI topologies. And sometimes we have to try different aggregates in order to find available resources, but recently, the online webpage [15], which provides status of resources at different aggregates, proved to be very helpful when reserving resources.

C. GENI Desktop

The GENI Desktop is a tool that can help visualize real-time measurements of GENI resources, which is useful when running experiments. The GENI Desktop provides built-in graphs for basic measurements, such as throughput, CPU usage, memory usage, IP/TCP/UDP traffic, *etc.* It also enables generating user-defined graphs, and provides interfaces that enable operations such as sending commands, uploading files and SSH to remote GENI resources. But the GENI Desktop has some limitations: (1) it does not support generating multiple data sets on the same graph, for example we cannot generate figures such as Figure 8; (2) users cannot fully control the appearance (font, scale, legend, axis label, *etc.*) of graphs; (3) there are lags (minutes) when generating real-time user-defined graphs; and (4) some functionalities (e.g., uploading files and SSH to VMs) are not stable and sometimes do not work.

VI. Conclusion and Future work

In this paper, we present how routing policies can be programmed using the RINA APIs provided by ProtoRINA. Through experiments running on the GENI testbed, we demonstrate how better video application performance can be achieved by choosing appropriate routing policies. We also report on our experiences with running GENI experiments over the past two years. We believe these experiences are helpful to both GENI experimenters and GENI developers.

We plan to have our ProtoRINA running on a long-lived slice over the GENI testbed, so researchers and educators can opt-in and benefit from the RINA architecture through programming recursive-networking policies (not limited to routing policies), leveraging the common mechanisms provided by ProtoRINA. We also plan to run larger-scale experiments that can stress-test both our RINA architecture and the GENI resources.

Acknowledgment

We would like to thank the National Science Foundation (NSF grant CNS-0963974) and the GENI Project Office. And thanks to Yue Zhu for his implementation of the video streaming application proxies.

References

[1] Open Networking Foundation White Paper, "Software-Defined Networking: The New Norm for Networks," April, 2012.
[2] J. Day, I. Matta, and K. Mattar, "Networking is IPC: A Guiding Principle to a Better Internet," in *Proceedings of ReArch'08 - Re-Architecting the Internet (co-located with CoNEXT)*, New York, NY, USA, 2008.
[3] Boston University RINA Lab, "http://csr.bu.edu/rina/."
[4] Y. Wang, F. Esposito, I. Matta, and J. Day, "RINA: An Architecture for Policy-Based Dynamic Service Management," in *Technical Report BUCS-TR-2013-014, Boston University*, 2013.
[5] Y. Wang, I. Matta, F. Esposito, and J. Day, "Introducing ProtoRINA: A Prototype for Programming Recursive-Networking Policies," *ACM SIGCOMM Computer Communication Review (CCR)*, July 2014.
[6] Y. Wang, F. Esposito, I. Matta, and J. Day, "Recursive InterNetworking Architecture (RINA) Boston University Prototype Programming Manual," in *Technical Report BUCS-TR-2013-013, Boston University*, 2013.
[7] ProtoRINA, "http://csr.bu.edu/rina/protorina/."
[8] GENI, "http://www.geni.net/."
[9] Y. Wang, F. Esposito, and I. Matta, "Demonstrating RINA Using the GENI Testbed," in *Proceedings of the Second GENI Research and Educational Experiment Workshop (GREE2013)*, Salt Lake City, UT, USA, March 2013.
[10] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A transport protocol for real-time applications," *RFC 3550*, July, 2003.
[11] Y. Wang, I. Matta, and N. Akhtar, "Experimenting with Routing Policies using ProtoRINA over GENI," in *Proceedings of the Third GENI Research and Educational Experiment Workshop (GREE2014)*, Atlanta, GA, USA, March 2014.
[12] GENI Portal, "https://portal.geni.net/."
[13] Y. Zhu, "RINA Video Streaming Application Proxies," https://github.com/yuezhu/rina_video_streaming.
[14] NetEm. Linux Foundation, "http://www.linuxfoundation.org/collaborate/workgroups/networking/netem."
[15] InstaGENI Rack Status, "http://groups.geni.net/geni/wiki/ExpGraphs."