PrimoGENI: Integrating Real-Time Network Simulation and Emulation in GENI

Nathanael Van Vorst, Miguel Erazo, and Jason Liu
School of Computing and Information Sciences
Florida International University
Emails: {*nvanv001, meraz001, liux*}*@cis.fiu.edu*

***Abstract*—The Global Environment for Network Innovations (GENI) is a community-driven research and development effort to build a collaborative and exploratory network experimentation platform—a "virtual laboratory" for the design, implementation and evaluation of future networks. The PrimoGENI project enables real-time network simulation by extending an existing network simulator to become part of the GENI federation to support large-scale experiments involving physical, simulated and emulated network entities. In this paper, we describe a novel design of PrimoGENI, which aims at supporting realistic, scalable, and flexible network experiments with real-time simulation and emulation capabilities. We present a flexible emulation infrastructure that allows both remote client machines and local cluster nodes running virtual machines to seamlessly interoperate with the simulated network running within a designated "slice" of resources. We show the results of our preliminary validation and performance studies to demonstrate the capabilities and limitations of our approach.**

***Index Terms*—network simulation, emulation, real-time simulation, virtualization**

I. INTRODUCTION

The success of Internet can be attributed to many factors. However, many argue that the very success has brought the ossification of its own design that makes it difficult to sometimes realize even the simplest changes. Political and economic factors notwithstanding, the success of transforming academic research to empirical implementation depends on the availability and trustworthiness of network testbeds to correctly expose important design and operational issues.

Considerable progress has been made in recent years in network testbed design to foster network innovations. The GENI initiative [1] capitalizes on the success of many previous efforts and strives to create a single collaborative and exploratory platform for future Internet research, bringing together the latest advances of network testbed research. The GENI design is built on four premises [2]:

1) *Programmability:* the ability to execute custom software on GENI-enabled facilities deep inside the network hierarchy;
2) *Resource sharing*: sharable resources among multiple users and experiments, possibly through virtualization;
3) *Federation*: interoperability among various resources provided by different organizations; and
4) *Slice-based experimentation*: supporting network experiments running independently within "slices" (a subset of GENI resources).

While GENI offers a cross-platform virtualization solution for physical and emulation testbeds, simulation is ostensibly missing in its original design. Although physical and emulation testbeds provide the necessary realism in prototyping and testing new network designs, simulation is more effective for studying complex behaviors of large-scale systems, which are both difficult to handle using realistic settings and intractable to close-form mathematical and analytical solutions.

PrimoGENI is a project that aims to incorporate real-time network simulation capabilities into the GENI "ecosystem". Real-time network simulation refers to simulation of potentially large-scale networks in real time so that the virtual network can interact with real implementations of network protocols, network services, and distributed applications. PrimoGENI uses our existing real-time network simulator, called PRIME [3], which is a real-time extension to the SSFNet simulator, capable of running on parallel and distributed machines for large-scale network simulations [4] . We augment PRIME with the GENI aggregate API, through which the experimenters can allocate resources and run experiments. PRIME can interoperate with the physical, simulated and emulated network components. In particular, network applications or protocols can run unmodified on emulated hosts, which are implemented as virtual machines, and interact with the simulated network. Network traffic generated by these applications or protocols is "conducted" on the simulated network operating in real time, with appropriate packet delays and losses calculated according to the simulated network conditions. In this respect, the simulated network is considered indistinguishable from a physical network.

In this paper, we present the design of PrimoGENI, which uses a current GENI control framework and extends the existing emulation infrastructure to support simulation and emulation experiments on a compute cluster. We highlight a set of major design decisions that direct our development and prototyping activities with the objective of substantially broadening GENI's capability in supporting realistic, scalable, and flexible experimental networking studies. Our contribution is three-fold:

1) PrimoGENI is the first of its kind to provide an end-to-end solution for building a real-time network simulation testbed that allows users to construct, deploy, and run network experiments involving physical, simulated and emulated network components in GENI.

978-1-4577-1366-8/11/$26.00

2) PrimoGENI features an infrastructure that allows multiple network experiments to run independently within their own slices, each possibly consisting of remote client machines and local cluster nodes running virtual machines that seamlessly interoperate with the simulated network.
3) We conducted preliminary validation and performance studies to demonstrate the capabilities and limitations of our approach.

The rest of the paper is organized as follows. In Section II we present the background and describe related work and the current GENI architecture. We describe in detail the PrimoGENI design in Section III. We present the result of our validation and performance studies of our preliminary implementation in Section IV. Finally we conclude this paper with a discussion of our ongoing and future work in Section V.

II. BACKGROUND

Network testbeds are commonly used for prototyping, evaluating, and analyzing new network designs and services. *Physical testbeds* (such as WAIL [5] and PlanetLab [6]) provide an iconic version of the network for experimental studies, sometimes even with live traffic. They provide a realistic testing environment for network applications, but with limited user controllability. Further, being shared facilities, they are overloaded due to heavy use, in which case it can severely affect their availability and accuracy [7]. An *emulation testbed* can be built on a variety of computing platforms, including dedicated compute clusters (such as ModelNet [8] and EmuLab [9]), distributed platforms (such as VINI [10]), and special programmable devices (such as ORL [11] and ORBIT [12]). While most emulation testbeds provide basic traffic "shaping" capabilities, *simulation testbeds* can generally achieve better flexibility and controllability. For example, ns-2 [13] features a rich collection of network algorithms and protocols that can be selected to model a myriad of network environments, wired or wireless. Simulation can also be scaled up to handle large-scale networks (e.g., SSFNet [4], GTNeTS [14] and ROSSNet [15]). It would be difficult and costly to build a large-scale physical or emulation testbed.

Real-time simulation is the technique of running network simulation in real time and thus can interact with real implementation of network applications [16]. Most existing real-time simulators (e.g., [17]–[20]) are based on existing network simulators extended with emulation capabilities. PRIME is a discrete-event simulator designed to run on parallel and distributed platforms and handle large-scale network models. PrimoGENI uses PRIME for real-time simulation, along with several important features:

- PRIME has an emulation infrastructure based on VPN that allows distributed client machines to remotely connect to the real-time simulator [21]. In a previous study, we conducted large-scale peer-to-peer content distribution network experiments using this emulation infrastructure [22].
- PRIME provides 14 TCP congestion control algorithms (mostly ported from the Linux TCP implementation), which have been validated carefully through extensive simulation and emulation studies [23].
- To allow large-scale traffic and reduce the computational requirement, PRIME applies multi-scale modeling techniques, allowing fluid traffic models to integrate with emulation traffic and achieving more than three orders of magnitude in performance over the traditional packet-oriented simulation [24].

GENI is a network research infrastructure that combines several prominent network testbeds, such as EMULAB [9], PlanetLab [6], and ORBIT [12], as a single collaborative and exploratory platform for implementing and testing new network designs and technologies. The GENI design consists of three main types of entities: clearinghouses, aggregates, and principals. A *clearinghouse* provides the central management of GENI resources for experimenters and administrators; specifically, it provides registry services for principals, slices and aggregates, and authentication services for accessing the resources. An *aggregate* represents a group of components encapsulating the GENI sharable resources (including computation, communication, measurement, and storage resources). When an experimenter from a research organization (i.e., a *principal*) decides to conduct a GENI experiment, she negotiates with the clearinghouse and the associated aggregate managers through an elaborate resource discovery and allocation process. In response to the experimenter's request, each participating aggregate allocates the requested resources for the experimenter, which are called *slivers*. Jointly, these slivers from all participating aggregates form a *slice*, which represents the set of allocated resources in which the experimenter will run network experiments.

A experimenter acquires resources from the GENI aggregates using a *control framework* run by the clearinghouse. GENI supports several control frameworks, including ProtoGENI [25], PlanetLab [6], ORBIT [12], and ORCA [26]. These control frameworks implement the GENI aggregate manager API that allows the experimenters with proper authentication to query for available sources, allocate resources upon the resource specification of an experiment, set up the resources for the experiment, and reclaim the resources after the experiment has finished. In particular, PrimoGENI extends the ProtoGENI control framework to manage, control and access the underlying resources. ProtoGENI is developed based on EmuLab [9], which is an emulation testbed implemented on a compute cluster. ProtoGENI extends EmuLab with key GENI functions, such as the registry services for principals, slices, and aggregate/component managers.

III. THE PRIMOGENI APPROACH

PrimoGENI runs a network model in real time, and thus can interact with real applications. Real-time simulation can provide accurate results since it is able to capture detailed packet-level transactions. It is flexible since one can easily explore the simulation model to answer what-if questions.

It also supports large-scale network experiments potentially with millions of simulated entities (hosts, routers, and links) and thousands of emulated hosts running unmodified network protocols and applications.

To interoperate with other GENI facilities, PrimoGENI functions as a GENI aggregate; as such, the experimenters can use the well-defined API to remotely control and realize network experiments consisting of physical, simulated and emulated network entities exchanging real network traffic.

A. Architecture

PrimoGENI provides an integrated development environment (IDE), called *slingshot*, for the experimenters to manage the "life cycle" of a network experiment, which includes model construction, configuration, resource specification, deployment, execution, monitoring and control, data collection, visualization and analysis. Slingshot provides a Python console for constructing and configuring network models interactively. One can also specify the network models using Java or XML. A network model describing the target system may consist of the topology of the network and a specification of simulated traffic on the network. One can also designate some network entities as *emulated hosts*, which are either virtual machines or physical machines with specified operating systems to run unmodified network protocols and applications. Slingshot provides a persistent storage for the network models using a database so that they can be inspected, analyzed and possibly reused after the experiment. Slingshot also provides a graphical representation of the network model for the experimenters to conveniently inspect and change the detailed configurations.

Once a network model is determined, the experimenter needs to specify the number of compute nodes to run the network experiment. Slingshot uses the METIS graph partitioner [27] to partition the network model by dividing the network model into subnetworks of approximately equal size and at the same time trying to maximize the communication latencies between the subnetworks (in order to achieve better lookahead for parallel simulation). Emulated hosts are given more weight in the network graph since it is important to spread the virtual machines evenly among the compute nodes as they require more resources. After that, the user can deploy and launch the experiment on a PrimoGENI cluster, which we describe in detail momentarily. During an experiment, the user can use slingshot to monitor the progression of the experiment and, if needed, change the state of the network entities when the experiment is running. The experiment results can be collected and displayed during or after the experiment. So far, we have implemented most of the aforementioned functions, except the online monitoring and control elements, which we leave for future work.

A PrimoGENI cluster consists of a set of compute nodes connected by one or more high-speed switches. PrimoGENI uses the ProtoGENI [25] control framework to manage, control and access the underlying resources. We designate one compute node to run the ProtoGENI aggregate manager and another one to run as the file server. The aggregate manager is responsible for allocating the resources, instantiating and setting up the virtual network, and connecting with the emulated hosts on the virtual machines and physical machines on the PrimoGENI cluster on behalf of the experimenter.

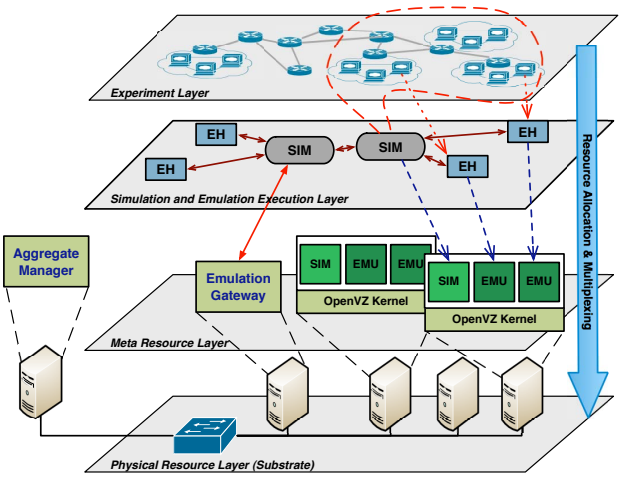

Fig. 1. PrimoGENI aggregate as a layered system

Our design makes a distinction between what we call meta resources and virtual resources. *Meta resources* include compute nodes and network connectivities between the compute nodes (such as a VLAN setup). We call them "meta resources" to separate them from the physical resources (which are known as the substrate). Meta resources also include virtual machines and/or virtual network tunnels. Meta resources are managed by ProtoGENI. PrimoGENI translates the experiment specification into meta resource requirements and instructs ProtoGENI to allocate the resources for the experiment.

Virtual resources are elements of the target virtual network, which include both simulated entities (such as hosts, routers and links), and emulated hosts. Emulated hosts can be implemented as virtual or physical machines, which run unmodified applications, such as web servers or software routers. These emulated hosts communicate with other hosts and routers on the virtual network (either simulated or emulated) according to the network model. Network traffic originated from or destined to an emulated hosts is called *emulated traffic*. All emulated traffic is "conducted" on the virtual network in simulation with proper packet delay and loss calculations as it competes with the simulated traffic.

The PrimoGENI aggregate can be viewed as a layered system, as illustrated in Fig. 1. At the lowest layer is the *physical resources (substrate) layer*, which is composed of compute nodes, switches, and other physical resources that constitute the EmuLab suite. These resources are made known to the clearinghouse(s) and can be queried by researchers during the resource discovery process. A compute node is set up to run the aggregate manager to export the aggregate API to experimenters and clearinghouses. PrimoGENI uses the ProtoGENI control framework, which provides common operations, such as creating, deleting and shutting down slivers. ProtoGENI uses secure socket layer (SSL) for authentication and encryption, and XML-RPC for remote procedure invocation.

The *meta-resource layer* is composed of a subset of resources allocated from the physical resource layer as needed by the experiment. After the requested resources have been successfully allocated by the ProtoGENI control framework, PrimoGENI will bootstrap each compute node by running a customized OpenVZ [28] OS image, which supports virtual machines. Each compute node will automatically start a daemon process, called the *meta-controller*, which awaits commands from slingshot to set up the simulation and emulation execution layer on behalf of the experimenter.

The *simulation and emulation execution layer* is created according to the network model of the experiment. Each compute node serves as a basic scaling unit and implements the operation of a subnetwork or a host that is mapped to it. A scaling unit for a subnetwork consists of a simulator instance, and zero or more emulated hosts each running as a virtual machine. The simulator instances on different scaling units synchronize and communicate with each other using parallel discrete-event simulation techniques. A scaling unit for a host runs the emulated host directly on the physical compute node. This is designed for cases where the emulated hosts have stronger resource requirements and thus cannot be run as virtual machines. PrimoGENI provides an emulation infrastructure to connect the emulated hosts running on virtual or physical machines to the simulator instances that contain the corresponding simulated hosts (see Section III-C). Each compute node runs a meta-controller. We designate one compute node to run the master meta-controller, which relays the commands from slingshot to the other (slave) meta-controllers for setting up the experiment (see Section III-D).

After the experiment starts, the experimenter can interact with the emulated hosts and the simulated network entities on the *experiment layer*. To access the emulated hosts, the experimenter can directly log onto the corresponding computing nodes or virtual machines and execute commands (e.g., pinging any virtual hosts). To access the dynamic state of the simulated network, PrimoGENI will provide an online monitoring and control framework (not yet implemented). This will allow the experimenter to query and visualize the state of the network. In situations where there is a large number of emulated hosts, one can also schedule slingshot commands to be executed at specific emulated hosts.

B. Compute Node Configuration

PrimoGENI uses the ProtoGENI control framework to manage the compute nodes in the cluster; ProtoGENI allows the compute nodes to be loaded with custom operating systems. To scale up emulation experiments, we use virtual machines (VMs) as emulated hosts.

Virtual machines come in different forms, but can be categorized generally as either system-level or OS-level VMs. System-level VMs can run complete guest operating systems (although sometimes with modifications), which can be different from the operating system running on the host machine. System-level VMs can provide strong resource isolation, but can support only limited number of VMs to run simultaneously with the guaranteed resources for each VM. OS-level VMs logically partition the systems into a set of distinct *containers*, each appearing to be a stand-alone machine. OS-level VMs share the same kernel as the host operating system. As such, they do not provide stringent resource isolation, but can support a large number of VMs to run on each physical host.

For PrimoGENI, we chose OpenVZ [28] to run the virtual machines for the emulated hosts where unmodified network applications can run. OpenVZ is an OS-level VM solution that meets our minimum requirement in terms of providing necessary separation of CPU, memory, and network resources among the VMs. CPU and memory separation is needed to limit the interactions of applications running on different VMs. Network separation is necessary to manage network traffic to and from the VMs each with an independent network stack. OpenVZ compartmentalizes the system resources inside the so-called containers; applications can run within these containers as if they were running on dedicated systems. Each container has its own set of processes, file system, and manages its own set of users (including root), and network stack (including network interfaces and routing/forwarding tables). With OpenVZ, PrimoGENI is able to perform network experiments with a large number of emulated hosts.

We preload the compute nodes with OpenVZ in the PrimoGENI cluster and provide an OS template from which the containers will be created. The OS template consists of a root file system and common network applications that may be run on a container (such as `iperf` and `tcpdump`). Additional applications can be added to individual containers using the meta-controller framework, which we describe in Section III-D. Since each container maintains its own file system, and the amount of storage space needed by the file system at each container may range from tens to hundreds of megabytes, which depends on the applications the experimenter wishes run; we would need a lot of space for running network experiments with a large number of emulated hosts. To solve this problem, we choose to use a union file system [29]. A union file system consists of two parts: a read-only file system (RF), which is common to all containers, and a writable file system (WF), which is specific to a container. A copy-on-write policy is used: when a file in RF needs to be modified, the file will be first copied to WF. To prepare the file system for each emulated host, PrimoGENI unions the default OS-template with an initially empty writable base file system. In this way we need only to store changes that later occur for each container as opposed to storing the entire file system.

C. Emulation Infrastructure

The emulation infrastructure in PrimoGENI needs to support high-throughput low-latency data communication between the emulated hosts and the simulation instances. There are two kinds of emulated hosts: collocated and remote. *Collocated emulated hosts* run as virtual machines on the same compute node as the simulator instance that simulates the subnetwork containing the corresponding virtual hosts. *Remote emulated hosts* are either physical compute nodes in the PrimoGENI

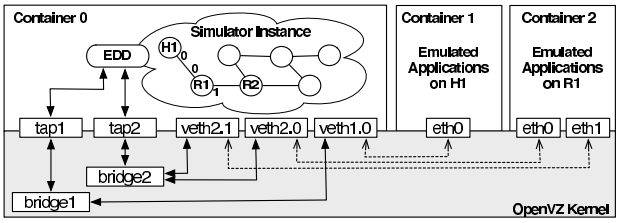

Fig. 2. An interconnection mechanism for collocated emulated hosts

cluster (not VMs), or machines that are not part of the PrimoGENI cluster and potentially at different geographic locations. Remote emulated hosts can run applications which otherwise cannot be run on a virtual machine due to either stringent resource requirements or system compatibility concerns. For example, an emulated host may require a special operating system (or version), or need specialized hardware that is not available on the compute node.

PrimoGENI's emulation infrastructure consists of three major components: the virtual network interfaces, the emulation device drivers, and the interconnection mechanism. The *virtual network interfaces (VNICs)* are installed at the emulated hosts, and treated as regular network devices, through which applications can send and receive network packets. Packets sent to a VNIC are transported to the simulator instance that handles the corresponding virtual host. The simulator subsequently simulates the packets traversing the virtual network as if they originated from the corresponding network interface of virtual host. Upon packets arriving at a network interface of an emulated virtual host in simulation, the packets are sent to the corresponding VNIC at the emulated host, so that its applications can receive the packets.

The *emulation device drivers (EDDs)* are collocated with the simulator instances running on the compute nodes. They are software components used by the real-time simulator to import real network packets sent from VNICs at the emulated hosts. Conversely, EDDs also export simulated packets and send them to the emulated hosts. Our emulation infrastructure supports different types of EDDs for remote or collocated emulated hosts. PRIME provides functions designed specifically for interactive simulations, which include handling real-time events and performing conversions between fully formed network packets and simulation events.

The *interconnection mechanism* connects VNICs on emulated hosts and EDDs with the simulator instances. Its design depends on whether it is dealing with remote or collocated emulated hosts. For remote emulated hosts, PrimoGENI uses the VPN-based emulation infrastructure, which we developed previously in PRIME [21]. In this scheme, OpenVPN clients are run at the the remote emulated hosts, each corresponding to one VNIC. PrimoGENI designates one or more compute nodes to run a modified OpenVPN server, which forwards IP packets to and from the EDDs collocated with the simulator instances. Detailed information about this interconnection mechanism can be found in [21].

Here we describe an interconnection mechanism designed for collocated emulated hosts, which uses Linux bridges and TAP devices. Fig. 2 depicts our design. For each collocated emulated host (an OpenVZ container), we create a software bridge and a TAP device connected to the bridge. Multiple bridges are used to segregate the traffic of each emulated host and ensure the emulated traffic is routed properly. In particular, we do not allow collocated emulated hosts to communicate with each other directly without going through the simulator. There are two possible alternatives. One could use VLANs to separate the traffic, although processing the VLAN headers may introduce additional overhead. One could also use ebtables, which is a mechanism for filtering traffic passing through a Linux bridge [30]. The TAP device is used by the EDD collocated at the simulator instance (in container 0) to send and receive packets to and from the VNICs[1]. The EDD also acts as an ARP proxy responding to the ARP requests from the emulated hosts, so as to direct packets originated from the VNICs to the associated TAP device.

The collocated emulated hosts are running as OpenVZ containers. We use a virtual Ethernet device [31] for each VNIC on the emulated hosts. The virtual Ethernet device is an Ethernet-like device, which actually consists of two Ethernet interfaces—one in container 0 (the privileged domain) and the other in the container where the emulated host is. The two interfaces are connected to each other, so that a packet sending to one interface will appear at the other interface. We connect the interface at container 0 to the bridge which corresponds to the emulated host.

We use an example to show how this mechanism works, by following the path of a ping packet from H1 via R1 and R2 shown in Fig. 2. Suppose both H1 and R1 are emulated in container 1 and 2, respectively. Before container 1 can send a packet to R1, assuming its ARP cache is currently empty, it sends an ARP request out from eth0 asking for the MAC address of R1's network interface eth0. The EDD responds with the MAC address of tap1, which is the TAP device connected to bridge1, the software bridge designated for container 1. Subsequently, container 1 is able to forward the ICMP packet to the EDD, which injects the packet into the simulator (by inserting an event that represents the packet sent out from network interface 0 of the simulated host H1). Once the simulation packet gets to R1 on the simulated network, the packet is exported from the simulator and sent by the EDD to eth0 in container 2 through tap2 and bridge2. Similarly, before container 2 forwards the packet onward to R2, it sends an ARP request out from eth1. The EDD responds with the MAC address of tap2, which is the TAP device connected to bridge2, the software bridge designated for container 2. In this way, the packet can find its way to the simulator, which forwards it on the virtual network.

[1]The latest TAP device implementation prohibits writing IP packets to the kernel, possibly for security reasons. We choose to use a raw socket instead for the EDD to send IP packets.

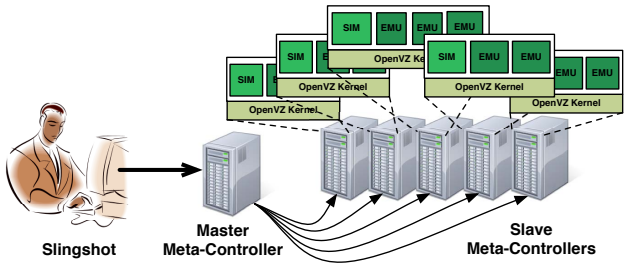

Fig. 3. Meta-controllers for experiment setup

D. Meta-Controllers

As mentioned earlier, PrimoGENI uses the ProtoGENI control framework to allocate the necessary compute nodes to run experiments. After the compute nodes are booted from the customized OpenVZ image, we need a mechanism to configure the compute nodes, create the containers, set up the emulation infrastructure, start the simulation, and launch the applications inside the containers. One can use XML-RPC for this purpose. However, since the commands are originated from slingshot, which is the user interface experimenters use to configure, launch and control the experiments, we would like to have a mechanism that allows the compute nodes to be coordinated locally on the PrimoGENI cluster to achieve better efficiency. In this section, we describe a *tiered* command framework, which we use to set up a network experiment and orchestrate experiment execution.

The command framework, as illustrated in Fig. 3, uses MINA, a Java framework for distributed applications [32]. MINA provides both TCP and UDP-based data transport services with SSL/TLS support. It is designed with an event-driven asynchronous API for high-performance and high-scalability network applications. Our implementation requires each compute node to run a meta-controller daemon process when first booted. When the meta-controller starts, it waits for an incoming connection. At this point the meta-controller takes no role, but after the connection is made, the meta-controller will become either a master or a slave. The difference between a master meta-controller and a slave meta-controller is that the master meta-controller is the one connected with slingshot; commands sent from slingshot, if they are not destined for the master, will be relayed to the corresponding slave meta-controllers. After the compute nodes have been allocated for an experiment, slingshot will choose one compute node to be the master and establish a secure connection to it using the experimenter's key obtained from the aggregate manager. After slingshot successfully connects to master meta-controller, slingshot instructs it to take control over the remaining compute nodes—the master establishes an secure connection to each of those compute nodes and instructs them to act as slaves.

After that, slingshot sends commands to the master meta-controller, which properly distributes the commands to the slave meta-controllers, if needed. Each command specifies the target compute node or one of its containers on the compute node, where the command is expected to run. A command can be either a blocking command or a nonblocking command. If it is a blocking command, the meta-controller will wait until the the command finishes execution and sends back the result of the command (i.e., the exit status) to slingshot. If the command is a nonblocking command, the meta-controller forks a separate process to handle the command and immediately responds to slingshot. Our current implementation uses blocking commands to set up the containers and the emulation infrastructure, and uses nonblocking commands to start the simulator and the user applications within the containers.

To set up the experiments correctly, the meta-controllers on the compute nodes need to run a series of commands:

1) *Set up MPI.* The master meta-controller creates the machine file and instructs the slave meta-controllers to generate the necessary keys for MPI to enable SSH logins without using passwords.
2) *Create containers.* The meta-controller creates a container for each collocated emulated host on the compute node. This step also includes creating the union file systems for the containers.
3) *Set up the emulation infrastructure.* For collocated emulated hosts, this step includes installing the virtual Ethernet devices in the containers, creating and configuring the software bridges and TAP devices, and then connecting the network devices to the bridges. For remote emulated hosts, this step includes setting up the OpenVPN server(s).
4) *Run the experiment.* The partitioned network model is distributed among the compute nodes. The master meta-controller initiates the MPI run, which starts the simulator instances on each compute node with the partitioned model.
5) *Start applications within containers.* Individual commands are sent to the meta-controllers to install and run applications at the emulated hosts.
6) *Shut down the experiment.* At any time, one can shut down the experiment by terminating the simulator, stopping the containers, and removing the emulation infrastructure (such as the bridges and the TAP devices).

All these commands are issued from slingshot automatically using the meta-controller command framework.

IV. EXPERIMENTS

In this section we describe the set of experiments we performed to validate the accuracy of our testbed, and determine its performance limitations, in order to show the utility of our approach. The experiments described in this section are conducted on a prototype PrimoGENI cluster with eight Dell PowerEdge R210 rack-mount servers, each with dual quad-core Xeon 2.8 GHz processors and 8 GB memory. The servers are connected using a gigabit switch.

A. Validation Studies

We validate the accuracy of the test by comparing the TCP performance between emulation and simulation. We use TCP for validation because it is highly sensitive to the delay jitters

and losses and therefore can magnify the errors introduced by the emulation infrastructure. Previously we performed validation studies for the VPN-based emulation infrastructure, which is designed for connecting applications running on remote machines [21]. In this study, we focus only on the emulation infrastructure based on software bridges and TAP devices, which we use to connect the collocated emulated hosts with the simulator instances run on multiple compute nodes.

We first compare the TCP congestion window trajectories achieved by the real Linux TCP implementations on the OpenVZ containers against those from our simulation. We arbitrarily choose three congestion control algorithms—BIC, HIGHSPEED, and RENO—out of the 14 TCP variants we have implemented in the PRIME simulator [23]. We use a dumbbell model for the experiments. The dumbbell model has two routers connected with a bottleneck link with 10 Mb/s bandwidth and 64 ms delay. We attach two hosts to the routers on either side using a link with 1 Gb/s bandwidth and negligible delay. The buffers in all network interfaces are set to be 64 KB. In the experiment, we direct a TCP flow from one host to the other that traverses the two routers and the bottleneck link. For each TCP algorithm we test three scenarios. In the first scenario, we perform pure simulation and use a simulated traffic generator. In the second and third scenario, we designate the two hosts as emulated hosts and use `iperf` to generate the TCP flow (and measure its performance). We run the emulation on one compute node for the second scenario, in which case the compute node runs simulator in container 0 and two other containers as the emulated hosts. In the third scenario, we use two compute nodes, each running one simulator instance and one emulated host. The emulated traffic in this case has to travel across the memory boundary between the two compute nodes in simulation (using MPI). For simulation, we use a script to analyze the trace output; for emulation, we sample the (`/proc/net/tcp`) file at regular intervals to extract the TCP congestion window size. Fig. 4 shows very similar TCP congestion window trajectories between simulation and emulation.

Next, we use a TCP fairness test to show whether our approach can correctly intermingle emulated and simulated packets. We use a similar dumbbell model; however, in this time, we attach two hosts on either side of the routers. We generate two TCP flows in the same direction—one for each of the two hosts on the left side to one of the two hosts on the right side. We select the TCP HIGHSPEED algorithm for both flows. We start one flow 20 seconds after the other flow. We compare two scenarios: in the first scenario we perform pure simulation; and in the second scenario, we designate the two end hosts of the first TCP flow as emulated hosts. The results are shown in Fig. 5. For both scenarios, we see that the congestion window size of the first TCP flow reduces when the second TCP flow starts; both flows eventually converge with a fair share of the bandwidth (at about 30 seconds after the second flow starts transmitting). Again, we see very similar results between simulation and emulation.

The emulation infrastructure inevitably puts a limit on the

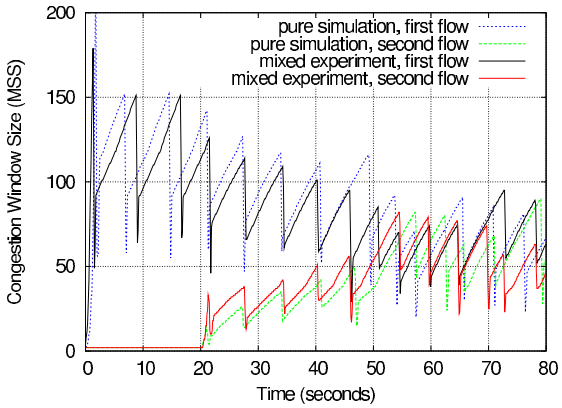

Fig. 5. Similar TCP fairness behavior between simulation and emulation

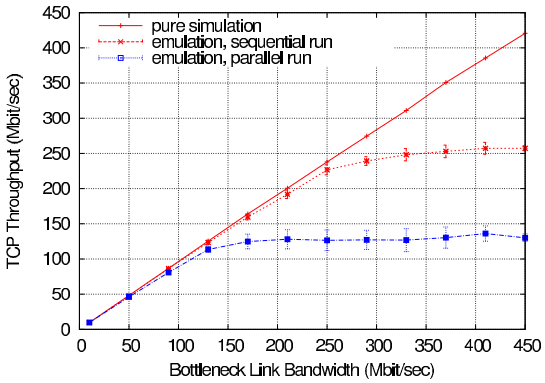

Fig. 6. TCP throughput limited by emulation infrastructure

throughput of the emulated traffic. In the last validation experiment, we look into this limiting effect on the emulated TCP behavior. Again, we use the dumbbell model. To increase the TCP throughput, we reduce the delay of the bottleneck link of the dumbbell model to 1 millisecond. For the experiment, we vary the bandwidth of the bottleneck link from 10 Mb/s to 450 Mb/s with increments of 40 Mb/s. Like in the first experiment, we direct a TCP flow from one host to the other through the bottleneck link and we compare three scenarios: the first using pure simulation, the second with an emulated flow within one compute node, and the third with an emulated flow across two compute nodes. Fig. 6 shows the results. While the throughput increases almost linearly with the increased bandwidth for the simulated flow, the error becomes apparent for emulated traffic at high traffic intensity. The throughput for the emulated traffic is kept below roughly 250 Mb/s for sequential runs and 130 Mb/s for parallel runs. The reduced throughput for parallel runs is due to the communication overhead as the TCP traffic gets exposed to the additional delay between the parallel simulator instances. Since emulation accuracy heavily depends on the capacity of the emulation infrastructure. We look into the emulation performance in more detail in the next section.

B. *Performance Studies*

In the previous experiment we see that the emulation infrastructure is placing an upper limit on the emulation traffic the system can support. Here we use a set of experiments

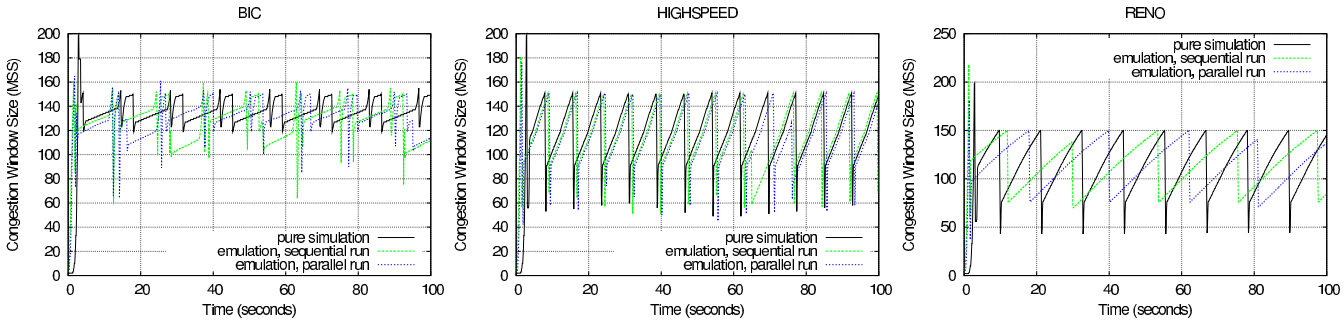

Fig. 4. Similar TCP congestion window trajectories between simulation and emulation

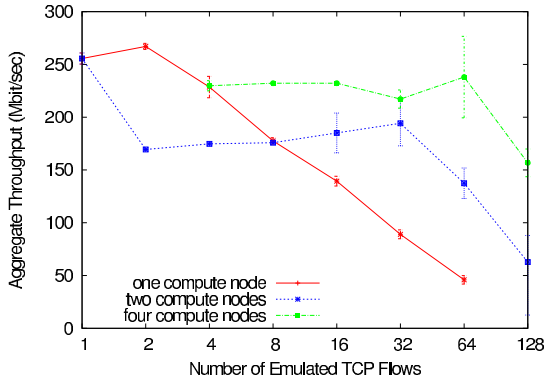

Fig. 7. Aggregate TCP throughput vs. emulated TCP flow count

to measure the capacity of the emulation infrastructure. We use the same dumbbell model (with a bottleneck link of 1 Gb/s bandwidth and 1 millisecond delay), and attach the same number of emulated hosts on each side of the dumbbell routers. We start a TCP flow (using iperf) for each pair of emulated hosts, one from each side of the dumbbell. So the total number of TCP flows is half the number of emulated hosts. We make same number of the TCP flows to go from left to right as those from right to left. We vary the number of emulated hosts in each experiment. We show the sum of the measured throughput (from iperf) for all emulated TCP flows in Fig. 7.

In the first experiment, we assign all emulated hosts to run on the same compute node. For one flow, the throughput reaches about 250 Mb/s, as we have observed in the previous experiment for sequential runs. The aggregate throughput increases slightly for two flows, but drops continuously as we increase the number of flows all the way to 64 flows (that's 128 VMs). The slight increase is probably due to TCP's opportunistic behavior that allows it achieve better channel utilization with more flows. We suspect that the drop in throughput is because the emulated hosts (OpenVZ containers) are competing for shared buffer space in the kernel network stack. With a smaller share when the number of containers gets bigger, the TCP performance degrades.

In the second experiment, we divide the network between two compute nodes (splitting the model along the bottleneck link). The throughput for one emulated flow in this case is around 130 Mb/s, like we have observed before. As we increase the number of flows, the aggregate throughput increases slightly until 32 flows, after which we start to see a significant drop. Running the simulation on two compute nodes results in more the event processing power, the simulator is thus capable of handling more emulated flows than the sequential case.

In the third experiment, we extend the dumbbell model by placing four core routers in a ring and connecting them using the bottleneck links. We attach the same number of emulated hosts to each core router. Each TCP flow was sent from an emulated host attached to one router to another emulated host attached to the next router in the ring. In this case, we spread the number of emulated flows evenly among the emulated hosts. Comparing with the scenarios with two compute nodes, the aggregate throughput for the four-node case is higher. Again we think is is due to the higher processing power of the parallel simulator. The throughput starts to drop for 128 flows (that's 32 VMs per compute node) as they compete for the shared buffer space.

V. SUMMARY AND FUTURE WORK

PrimoGENI supports real-time network simulation and emulation on GENI enabled resources. It provides the necessary tools for the experimenters to construct and configure the network models, allocate the resources, deploy and run the experiments, and collect the experiment results. Each experiment is run within its own slice of resources, allocated using the GENI control framework. The resources may consist of the compute nodes in the cluster and possibly remote machines. Each compute node is treated as a scaling unit in PrimoGENI; it can be configured to run a parallel simulator instance together with a set of virtual machines for emulated hosts. The latter provides a realistic operating environment for testing real implementations of network applications and services. PrimoGENI applies parallel and distributed simulation techniques to synchronize the simulator instances running within the scaling units. The virtual machines are connected with the simulator instances using a flexible emulation infrastructure.

Our validation experiments show that PrimoGENI can produce accurate emulation results when the emulated traffic does not exceed the capacity of the emulation infrastructure. Our performance studies show that the throughput of the emulated traffic is dependent upon the number of virtual machines running on each scaling unit.

We are currently investigating more efficient mechanisms to increase the emulation throughput. For example, we want

to apply zero-copy techniques for moving data packets more efficiently between the virtual machines. We also want to have a fine-grained control of the I/O functions inside the real-time simulator to reduce the overhead. Another technique is to slow down the emulation speed using time dilation [33], which controls a systems notion of how time progresses on the virtual machines. This is achieved by enlarging the interval between timer interrupts by what is called a time dilation factor (TDF), thus slowing down the system clock. If we proportionally slow down the real-time simulator, the applications can experience a higher networking speed than what can be offered by the emulation system.

We will extend PrimoGENI to allow the virtual network in a slice to be able to connect to other GENI resources to facilitate larger network experiments. We will also implement the mechanism to control, inspect, and visualize the state of the network experiment. To access and modify the runtime state, we need to implement a dynamic query architecture based on the meta-controller design. Our design will consist of three components: a database that runs separately from the simulator, the data managers for collecting the states in simulation, and the query gateways which ferry query requests and results between the database and the data mangers. We will investigate strategies for complex queries involving collectieve operations on the large network state.

ACKNOWLEDGMENTS

We would like to thank our undergraduate students, Eduardo Tibau and Eduardo Peña, for the development of the slingshot IDE. The PrimoGENI project is part of GENI's Spiral 2 prototyping and development effort, funded by the National Science Foundation (NSF) through GENI Project Office (GPO). The PrimoGENI project is built on PRIME, which is also funded by NSF through the grant CNS-0836408.

REFERENCES

[1] GENI, http://groups.geni.net/.
[2] GENI System Overview, http://groups.geni.net/geni/wiki/GeniSysOvrvw.
[3] PRIME, http://www.primessf.net/.
[4] J. Cowie, D. Nicol, and A. Ogielski, "Modeling the global Internet," *Computing in Science and Engineering*, vol. 1, no. 1, pp. 42–50, 1999.
[5] P. Barford and L. Landweber, "Bench-style network research in an Internet instance laboratory," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 21–26, 2003.
[6] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A blueprint for introducing disruptive technology into the Internet," in *Proceedings of the 1st Workshop on Hot Topics in Networking (HotNets-I)*, 2002.
[7] N. Spring, L. Peterson, A. Bavier, and V. Pai, "Using PlanetLab for network research: myths, realities, and best practices," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 1, pp. 17–24, 2006.
[8] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker, "Scalability and accuracy in a large scale network emulator," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, 2002, pp. 271–284.
[9] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, 2002, pp. 255–270.
[10] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford, "In VINI veritas: realistic and controlled network experimentation," in *Proceedings of the 2006 ACM SIGCOMM Conference*, 2006, pp. 3–14.
[11] J. DeHart, F. Kuhns, J. Parwatikar, J. Turner, C. Wiseman, and K. Wong, "The open network laboratory," *ACM SIGCSE Bulletin*, vol. 38, no. 1, pp. 107–111, 2006.
[12] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh, "Overview of the ORBIT radio grid testbed for evaluation of next-generation wireless network protocols," in *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC 2005)*, 2005.
[13] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu, "Advances in network simulation," *IEEE Computer*, vol. 33, no. 5, pp. 59–67, 2000.
[14] G. F. Riley, "The Georgia Tech network simulator," in *Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research (MoMeTools'03)*, August 2003, pp. 5–12.
[15] G. Yaun, D. Bauer, H. Bhutada, C. Carothers, M. Yuksel, and S. Kalyanaraman, "Large-scale network simulation techniques: examples of TCP and OSPF models," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 27–41, 2003.
[16] J. Liu, "A primer for real-time simulation of large-scale networks," in *Proceedings of the 41st Annual Simulation Symposium (ANSS'08)*, 2008, pp. 85–94.
[17] K. Fall, "Network emulation in the Vint/NS simulator," in *Proceedings of the 4th IEEE Symposium on Computers and Communications (ISCC'99)*, 1999, pp. 244–250.
[18] R. Simmonds and B. W. Unger, "Towards scalable network emulation," *Computer Communications*, vol. 26, no. 3, pp. 264–277, 2003.
[19] X. Liu, H. Xia, and A. A. Chien, "Network emulation tools for modeling grid behavior," in *Proceedings of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'03)*, 2003.
[20] J. Zhou, Z. Ji, M. Takai, and R. Bagrodia, "MAYA: integrating hybrid network modeling to the physical world," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 14, no. 2, pp. 149–169, 2004.
[21] J. Liu, Y. Li, N. V. Vorst, S. Mann, and K. Hellman, *J. Systems & Software*, vol. 82, no. 3, pp. 473–485, 2009.
[22] J. Liu, Y. Li, and Y. He, "A large-scale real-time network simulation study using PRIME," in *Proceedings of the 2009 Winter Simulation Conference*, December 2009.
[23] M. Erazo, Y. Li, and J. Liu, "SVEET! A scalable virtualized evaluation environment for TCP," in *Proceedings of the 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom'09)*, April 2009.
[24] J. Liu and Y. Li, "On the performance of a hybrid network traffic model," *Simulation Modelling Practice and Theory*, vol. 16, no. 6, pp. 656–669, 2008.
[25] ProtoGENI, http://www.protogeni.net/.
[26] ORCA, https://geni-orca.renci.org/trac/.
[27] METIS, http://www.cs.umn.edu/~metis/.
[28] OpenVZ, http://wiki.openvz.org.
[29] FunionFS, http://funionfs.apiou.org/.
[30] ebtables, http://ebtables.sourceforge.net/.
[31] OpenVZ Virtual Ethernet Device, http://wiki.openvz.org/Veth.
[32] Apache MINA, http://mina.apache.org/.
[33] D. Gupta, K. Yocum, M. McNett, A. Snoeren, A. Vahdat, and G. Voelker, "To infinity and beyond: time-warped network emulation," in *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'06)*, 2006.