Qemunet: an Approach to an Automated Virtualized Testbed

Pavel Boyko
Institute for Information Transmission Problems
19 Bolshoy Karetny per.
Moscow, Russia 127994
boyko@iitp.ru

Andrey Mazo
Institute for Information Transmission Problems
19 Bolshoy Karetny per.
Moscow, Russia 127994
mazo@iitp.ru

ABSTRACT

This paper presents a software system that allows for multiple virtual machines connected by a network simulator to be utilized as an integrated network testbed. The system is built upon popular network simulator ns-3 and is capable of synchronizing user-supplied commands across different virtual machine instances. Performance evaluations are given to estimate limits of applicability of the system.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design - Wireless Communication

Keywords

Testbed, ns-3, virtual machine, emulation

1. INTRODUCTION AND MOTIVATION

Sooner or later development of any viable network protocol reaches a stage of testbed-based evaluation. Testbeds represent an important step in developing real world network protocols, and without them such development would be far from optimal. Constructing an own small testbed is not an easy task for protocol developers while own large testbeds are completely unachievable by them. Though there are many publicly available testbeds of various sizes and characteristics, they are still hard to use and have different limitations. In addition, the requirement to share a public testbed with others is not suitable at all times.

The main problems preventing developers from deploying their own testbeds are lack of available hardware and impossibility to manage desired links between nodes. These two problems can be effectively solved by taking advantage of virtualization technologies and network simulators. Such solutions allow a developer to get a whole network of (virtual) stations running given just a couple of wide-spread desktops or servers. Although a virtual network can be configured and launched manually, this routine quickly becomes tedious and eventually evolves in at least a set of ad-hoc automated scripts. There exists a number of projects to develop comprehensive real-time network emulators based on symbiosis of virtualization systems and network simulators. Most of them handle well an initial setup, launch and termination of virtual stations, creation and control of network topologies. But most of them fail to provide a user with sufficient support to conduct complicated experiments in automated manner. For example, a user could hardly perform experiment-specific actions exactly in time.

The main purpose of this work is to develop a system not only capable of creating and interconnecting virtual stations but also capable of synchronizing user-specific events between different virtual stations. Such a system readily enables long complicated experiments with branching sequences of commands to be conducted automatically. Otherwise, either a task of performing hundreds of experimental runs looks completely impossible or they may experience random run time failures due to unexpected events ordering.

Our first idea was to use ns-3 simulator as an external tool, similar to how Network Experiment Programming Interface (NEPI) (see subsection 2.4) does this. Ns-3 was intended to replace a simple bridging solution or a more flexible solution based on Virtual Distributed Ethernet (VDE) (see subsection 2.5). The prototype system was written in Perl. But event synchronization problems raised very early. One can start the actions aimed at the actual testing (such as measuring bandwidth) only after all nodes are up and configured (for example, IP addresses are assigned to them). There were some ad-hoc, proof-of-concept synchronization solutions, that required sometimes non-trivial support from ns-3 side. After that a decision was made to integrate the whole system into ns-3 framework. This ultimately led to form the Qemunet system described in this work.

The remainder of the paper is organized as follows. First, we discuss some existing technologies, that we will make use of, and solutions that aim at similar tasks in section 2. Then we describe a proposed solution and implementation architecture in section 3. After that, in section 4 we show some performance measurements to assess the domains of applicability for our system. And in section 5 we summarize the efforts and give directions to further work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WNS-3 2011, March 21, Barcelona, Spain
Copyright © 2011 ICST 978-1-936968-00-8
DOI 10.4108/icst.simutools.2011.245559

2. RELATED WORK

2.1 Virtualization

Virtualization means the creation of multiple isolated instances of a piece of software running on a single host plat-

form. Virtualization techniques can be classified by the degree of abstraction and therefore by the nature of programs being isolated from each other. We will now describe in short all of the most wide-spread techniques.

Full virtualization (Virtual Machine).

A virtual machine performing full virtualization fully emulates all required hardware with all its peculiarities and quirks. Full virtualization is completely transparent to software being virtualized. So any operating system (OS) supporting particular set of emulated hardware can be run inside virtual machine with no modifications. An interesting feature of full virtualization is that it allows running software designed for one hardware architecture on an entirely different hardware.

We are mostly interested in full virtualization systems because they allow running guest OSes and applications unmodified. For example, it looks very attractive to be able to take full file system image dump from a real wireless router and instantiate tens of router clones in a single simulation with minimal modifications to the image itself. Also, this can be an useful approach to evaluation and validation of a protocol prototype implementation.

The main drawback of full virtualization systems is performance issues. Due to its low-level and high-detailed nature, full virtualization requires more host machine resources than other virtualization approaches. But recent advances in hardware virtualization technologies such as Intel VT-x or AMD-V made full virtualization much simpler to implement for widespread x86 architecture and raised the performance much closer to native execution level.

Examples of popular products providing full virtualization are Qemu [1], VMware [2], VirtualBox [3].

Paravirtualization.

Like full virtualization paravirtualization makes it possible to run full guest OS independently of another running guest OSes or other processes. But to reduce performance penalties of full virtualization, paravirtualization requires guest OS to be modified to avoid usage of hard-to-virtualize processor instructions as much as possible. In other words, guest OS “knows” that it is running inside host OS. This allows for the execution of most of guest OS code with the near to native performance. But the major disadvantage is that guest OS is bound to particular available hardware. So it is impossible to run code generated for a different architecture.

Xen [4] and User Mode Linux [5] are examples of paravirtualization products.

OS-level virtualization.

Many developed programs do not require exclusive direct hardware access. For example, user-space routing daemons operate with network devices through a high-level abstraction of sockets. Such programs often require just minimum (or not at all) isolation from other similar programs. In such a case, OS-level virtualization is the best choice due to its unnoticeable performance degradation. For this type of virtualization OS simply creates multiple copies of otherwise unique system resources, e.g. global routing table, file system, network interfaces. Unfortunately, such OS techniques are hardly portable across different operating systems. Of course, no hardware or foreign architecture emulation is possible for OS-level virtualization. OpenVZ [6], Linux Containers (LXC) [7] and FreeBSD Jail [8] can be named among other implementations.

At the beginning of Qemunet development, the ability to run unmodified software and software for diverse hardware platforms during experiments was crucial. Therefore, originally Qemunet system was built with full virtualization solutions in mind and was particularly adapted to work with Qemu VM. However, Qemunet could be easily extended to support other virtualization products.

2.2 Discrete-event realtime simulation

Emulating non-trivial behavior of physical and data link layers of OSI model requires quite complex models. Developing such models requires much time and effort. Reusing already developed, tested and validated models is a logical and sound solution. There are several popular network emulators providing a set of reusable models. As virtualized emulated nodes run in realtime, corresponding network simulator models have to be computable with hard realtime limits too. The great advantage of ns-3 [9] is that it is capable of running in realtime and also has an easy-to-use mechanism for exchanging packets with a host system. In addition to intelligent mechanism for model and realtime clock synchronization, there is a built-in support to control inaccuracies in events’ dispatching time.

2.3 CORE

Common Open Research Emulator (CORE) [10, 11] aims at the similar task as Qemunet does. But it exploits other means to achieve the same goal. CORE is capable to instantiate a large number of lightweight virtual machines (actually, OS-level virtualization based) and interconnect them with links of different types emulating either behavior of wired or wireless connections. CORE was initially designed to make use of fast in-kernel facilities to forward and influence packets from one virtual machine instance to another. It’s possible to emulate wired links with given throughput, delay and packet loss ratio completely in host OS kernel. The requirement to account for such complex effects as RF interference and packet collisions in wireless channels imposed the need to separate some computationally complex modules into a user-space daemon. CORE is also featuring a nice looking GUI to control network topology in real time. CORE is built on a modular architecture and there are various modules allowing it to be run on various OSes and to use various network simulators. Until recently, CORE has no support for creating virtual networks with ns-3 simulator and thus has a relatively limited set of models. Also, CORE provides a mechanism to execute commands on virtual nodes but provides no synchronization between them.

2.4 NEPI

Main task of Network Experiment Programming Interface (NEPI) [12, 13] is to simplify and unify diverse configuration interfaces provided by independent emulation projects. Thus NEPI allows easy mixing of different simulators, emulators and testbeds to perform an experiment. It features considerable object model, that is abstract enough to meet different emulator needs while still being quite intuitive. At the moment, NEPI can interact with at least ns-3 simulator and a special testbed project designed for configuring Linux-

based OS-level virtualization and Linux-based network phenomena emulation. One of the promising features of NEPI about to be finished is an ability to establish necessary tunnels to remote testbed machines to transparently connect them to the experiment being held. Nevertheless, for now, NEPI has a limited view of an application running during an experiment. I.e. NEPI considers an application as a single entity, that does not require external synchronizing supervisor. Thus the experiment designer has to utilize some improvised synchronization technique.

2.5 VDE

Virtual Distributed Ethernet (VDE) [14] is a generic solution for interconnecting network interfaces and creating virtual topologies. It is capable of operating any network interfaces through general enough approach of reading, modifying and writing packet captures via libpcap [15]. VDE has ability to be spawned across multiple physical machines. It has extensible design and therefore many adapters to connect to well-known virtualization projects like Qemu [1] or UML [5]. The major drawback of VDE is that it is hard to impossible to model wireless MAC and PHY phenomena using its architecture.

3. QEMUNET ARCHITECTURE

Qemunet is fully integrated into ns-3. Thus all ns-3 facilities and frameworks are available for use in Qemunet. The ones most widely used in Qemunet include realtime event scheduler, attribute system, synchronization and threading primitives, callbacks. The major part of Qemunet executes inside the main simulator thread, in which other viable parts of the simulator run. This results in a limitation that no function calls can be allowed to be blocking ones. Avoiding synchronous function calls made decisive impact on Qemunet architecture and heavy use of callbacks.

In addition to the main simulator thread there are also some specific components running in separate threads. All interaction between threads is carried out through thread-safe realtime scheduler [16].

It's important to note that Qemunet is not just another solution to automatically launch and interconnect virtual machine based nodes with high fidelity physical and data link layer models provided by a network simulator. But the key feature, that distinguishes Qemunet from projects like CORE (see subsection 2.3), NEPI (see subsection 2.4) or VDE (see subsection 2.5), is the possibility to execute user commands simultaneously on several controlled nodes. Moreover ns-3 is an integral part of Qemunet because it is heavily used for synchronization between different threads controlling different virtual machines. Ns-3 also acts as wall clock for time measurements throughout the system. With Qemunet it is easy to perform experiments with high confidence level, because thousands of independent runs may be performed automatically with no human control. While composing scenarios you can easily set which commands must be executed to configure a particular node, or to start actual experiment right after all "qemu" nodes became ready, or execute commands involving multiple nodes in the right order with no guard time intervals. For example, you do not need to insert random number of `sleep(1)` or `sleep(100)` calls in your script just to be sure, that everything is already up and ready to start actual measurements.

Experiment's main part (besides configuration) is usually represented by running applications on multiple nodes. Such applications can consist of a single entity or several united entities. For example, running `iperf`-like application to measure bandwidth and/or packet loss means running a single entity application. On the contrary, an application may represent a sequence of actions. For example, the sequence "run iperf, then check routing tables, then check traceroute output" can also be thought of as an application. But this application actually consists of several individual commands. Such a sequence can be emulated via launching all the required commands at once but with a specific delay incorporated into each one. Most emulation and testbed control frameworks only have a notion of an application, but have no idea of individual commands. This works well in many cases as no precise synchronization between commands simultaneously executed on different virtual machines is really required. Or there are usually at least some rough assumptions on the duration of individual commands. Applications could rely on `sleep(1)` for primitive synchronization. But with complex experiments, compound of multiple commands, it may not be an easy task to place all the necessary delays in the right places. Because each application runs on a separate node, it's quite a tricky task to develop command synchronization solution, that would not interfere with the experiment itself. So the major goal of Qemunet is to introduce idea of commands and well defined stages of an experiment run. Basic concepts of Qemunet are a *command* and a *group of commands* or a *Task*.

The overall process of Qemunet initialization looks like:

1. read and parse topology description;
2. create, connect and configure ns-3 nodes based on the given topology;
3. for each "qemu" node spawn separate Qemu VM with a number of network interfaces according to the given topology;
4. setup TapBridge on every network interface of a given "qemu" node inside ns-3;
5. connect Qemu's host OS Tap interface with ns-3 Tap interface created by TapBridge;
6. wait for guest OS inside VM to boot and perform some initial configuration;
7. start executing user tasks (see below).

A typical Qemunet experiment consists of:

- initial network topology description (with nodes' types, positions, interfaces, links, IP addresses, etc.);
- usual ns-3 C++ scenario, that assigns commands to nodes and group commands to be started simultaneously;
- some ad-hoc script (or a full-blown system) to gather scenario's output, control number of runs and to do the search over an input parameter space.

All major Qemunet components with important interactions between them are depicted in Figure 1. Now we'll explicitly list and then describe them in detail:

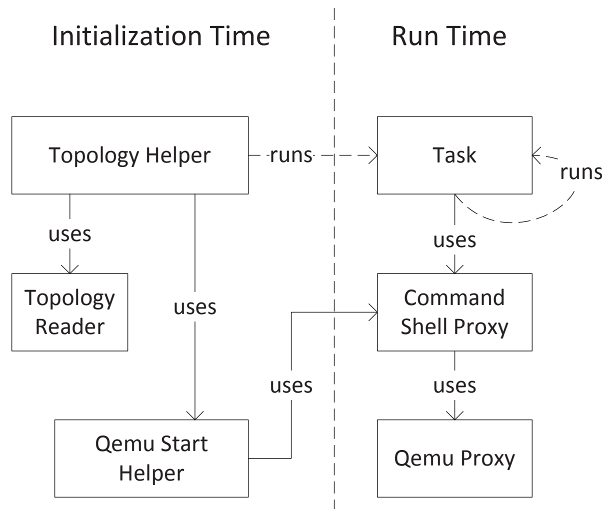

Figure 1: Qemunet component interaction

Topology Reader parses topology description and converts it to internal representation;

Topology Helper creates, configures and interconnects nodes based on internal topology representation;

Qemu Start Helper creates Qemu instances with care using parts of internal topology representation;

Qemu Proxy is a proxy to a remote Qemu process with exported standard input and output;

Command Shell Proxy capable of sending commands to a process via child's standard input and receiving output via child's standard output;

Task group and synchronize commands to be executed simultaneously on different nodes.

As it can be seen from the list of Qemunet components, all Qemu-specific interaction is encapsulated into *Qemu Start Helper* and *Qemu Proxy*. It makes possible to add support for other virtualization solutions like LXC or OpenVZ. They just have to be smart enough to support command-line configuration, instantiation and termination. However, replacing ns-3 with another network emulation tool seems to be unfeasible idea.

One component, that isn't strongly related to Qemunet but that we should mention, is an ad-hoc script intended to clean up the host system after dirty termination of Qemunet based simulation.

Topology Reader.

Creating relatively large and/or complex topologies via hand-written ns-3 C++ code poses significant difficulties and is quite error-prone. A visual tool for creating topologies has recently appeared [17]. Unfortunately, we had started the work on Qemunet before that tool was even announced publicly. While generating C++ scenario code with a third party tool is a viable solution, it is quite sensitive to version mismatch between the tool and ns-3 due to various API changes. We took another approach such as describing a topology in a simple (though extensible) ad-hoc text format. So there is a cleanly defined format to specify a topology, which allows generator tools and ns-3 to evolve independently. On ns-3 side there must be a special topology reader module that is able to parse the topology description format and convert it to some internal topology representation.

Topology Helper.

This is one of the most important parts of Qemunet. Given topology description from *Topology Reader*, *Topology Helper* performs necessary ns-3 function calls to create and interconnect ns-3 nodes and setup applications on them. To support non-trivial node configurations and to keep topology format simple, we added notion of *node type*. The examples of node type are "network bridge" or "OLSR equipped router", that tells *Topology Helper* to create an additional bridge interface on a given node or to install OLSR application on it respectively. In this paper we focus on describing the "qemu" node type. For each "qemu" node *Topology Helper* must instantiate Qemu virtual machine and "attach" it to the node. After a qemu is attached to a node it fully controls the behavior of the node, it receives and transmits packets on behalf of the node. For any "qemu" node the corresponding ns-3 node requires just minimal configurations while the most complicated configuration is performed in Qemu's guest operating system. Configuration of pure ns-3 nodes (like applications or Internet stack installation) is done via ordinary ns-3 functions and does not differ much from a usual ns-3 scenario. But configuration of ns-3 nodes of type "qemu" can't be achieved in such a smooth way and thus is split out to a separate component: *Qemu Start Helper* (see below). This is the *Topology Helper*'s responsibility to decide, whether the virtual testbed is prepared, and to start the actual experiment by transfering control to a user defined sequence of commands.

Qemu Start Helper.

Running and controlling external programs from ns-3 poses several difficulties. This is caused mostly because C/C++ standard library doesn't provide comprehensive and easy-to-use API for talking to external processes. For example, Python provides nice Subprocess module [18], or there is IPC::Run [19] module for Perl. TCL Expect [20] module can also be named as such an easy-to-use IPC module. Ns-3 is bundled with Python bindings which are close enough to the corresponding C++ API, but they have particular compatibility problems with realtime scheduler [21] and for now all Qemunet components are written solely in C++. More specifically we can distinguish the following problems when dealing with external processes (with no particular order):

1. spawning a process with correct command-line arguments and environment;
2. talking to and listening to the process via pipes or so;
3. handling exceptions in the spawned process;
4. correct termination of the spawned process.

Qemu Start Helper's role is to prepare all the required parameters and files for correct Qemu invocation. To properly start Qemu, one must decide on exact command-line

switches and must prepare binary images of a guest OS kernel and a root file system. Allowing for controlling information exchange between the main simulator process and Qemu processes is a joint task of *Qemu Start Helper* and *Qemu Proxy*. While *Qemu Proxy* handles low-level details such as pipes creation, Qemu Start Helper is responsible for tuning Qemu via command-line switches to tell the Qemu process to use these pipes for information input-output. Qemu command-line parameters controlled by *Qemu Start Helper* can be divided into two classes: invariable and dependent on the given network topology. Invariable parameters are usually the minimum required parameters to properly start Qemu VM and to establish controlling channel to it. Parameters dependent on the network topology are, for example, ones that specify number of network interfaces inside VM and their hardware MAC addresses.

Guest OS running inside VM usually is not very useful without some initial configuration such as IP addresses assignment. As *Qemu Start Helper* is already aware of all network interfaces on a specific Qemu node, it is the right place to perform initial configuration after guest OS boot. In addition to IP addresses configuration, one may want to set a host name for the guest OS based on information provided from topology description. Or it's reasonable to launch some routing daemons with some node-specific information as parameters. There are a lot of tools that are meant to configure network interfaces, which may or may not be available in a given OS distribution (e.g., Linux distribution). To support such variety of guest OSes, *Qemu Start Helper* provides a special callback for such initial guest OS configuration, that can be overwritten with a user supplied configuration. With the help of the callback, *Qemu Start Helper* makes up a (potentially long) text command to configure a particular guest OS.

In addition to controlling information exchange between the main simulator process and a Qemu process, there must be some way to send packets between the simulator and guest OS back and forth. Given the fact that both Qemu and ns-3 are capable of exporting Tap interface to host OS side, the solution for this task looks trivial. One just have to forward packets from Qemu's Tap device to ns-3 Tap device on host OS side, which is an easy task with several available open-source solutions. The solution that is ready to be used out-of-the box on most Linux distributions is a kernel-side bridge. Because creating and connecting various Tap devices must be done on per-node basis and also involves the knowledge of node's addresses, *Qemu Start Helper* fits the role best. So, *Qemu Start Helper* must:

- create TapBridge NetDevice over the right ns-3 node's interface;
- choose correct parameters for Qemu invocation for it to create Tap device for right guest OS interface;
- create and configure host OS bridge to interconnect the two provided Tap devices (one for each ns-3 node's network device and one for each guest OS's network device).

Qemu Proxy.

Qemu is running as a separate process and does not support C++ API required for other Qemunet components. So there is a requirement for a special proxy object, accessible from other ns-3 objects and representing Qemu inside ns-3. *Qemu Proxy* is the object.

In short, *Qemu Proxy* must:

- launch Qemu VM on instantiation;
- stop Qemu VM on destruction;
- create bi-directional control channel to Qemu process.

Qemu Proxy is a low-level class, it is not affected, for example, by whether the Qemu command line parameters are correct or not. *Qemu Proxy*'s task is to spawn Qemu process on its instantiation and to terminate the process on its destruction. Just spawning Qemu process is not very helpful as it doesn't allow any information exchange between the main simulator process and the Qemu process. So *Qemu Proxy* ought to establish a bi-directional channel to each Qemu process. This is done via two standard uni-directional POSIX pipes. This is transparent to Qemu and it behaves as if it were interacting with a human via standard input-output. Termination of a VM is implemented via simple kill of a corresponding Qemu process. This allows for freezing a guest file system in a desired state. Fatal exceptions in spawned Qemu processes are not handled properly at the moment. Nevertheless, they will most likely lead to an abnormal termination of the ongoing simulation and thus will not be silently ignored.

How can we control any processes running inside Qemu VM? It is possible to launch a special daemon inside Qemu, that will understand commands from the main simulator process and perform the needed operations to fulfill such requests. Such approach restricts potential users only to commands supported by such a daemon and appropriate Qemunet module. Also, this approach means reimplementing functionality from many well-established system utilities. At the same time, well-known *nix commands are easy to use, stable enough, well documented and familiar to even unexperienced users. So the chosen solution is based on sending usual text commands to guest OS's command shell (e.g., Bash or Ash) and then collecting and reporting the output to the user. *Therefore, the most severest restriction for a guest OS is to provide a command shell on its first serial console.* In addition to the required serial console, the guest OS can provide other consoles and a graphical window system, that can be used by a user to create or debug sequence of commands. Command line interfaces are usually human oriented and as such pose several difficulties to an automated control system. Sending commands in time, collecting command outputs are not so trivial tasks, so a special component, *Command Shell Proxy*, is required to deal with a command shell.

Command Shell Proxy.

From a user point of view a command shell is a convenient way to deal with guest OS. For a software console output parser there are several difficulties. Console output parser has no information about what a particular command should do and what its output might look like. Completion of a command is determined via shell prompt appearing on standard output after command invocation. So an important assumption is the presence of command shell prompt, that is unchanged throughout the guest OS existence. Also, it should be noted, that the command prompt must be quite unique not to accidentally coincide with some command's

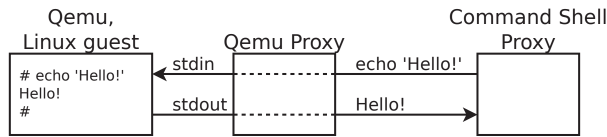

Figure 2: *Qemu*, *Qemu Proxy* and *Command Shell Proxy* relationship

output (such a situation will result in lost synchronization and thus in unexpected consequences). Also, any use of interactive programs is not allowed, because all of them use different prompts and workflows and it becomes a substantial workload to support all of them in a single system. Fortunately, many wide-spread system tools are non-interactive, so the demand on non-interactiveness is not very restrictive.

Sending a command to a guest OS is quite straightforward. Instead, receiving the command's output is not so easy. Guest OS may return output with differently sized parts. Also, it may produce some output even if no command has been entered. So *Command Shell Proxy* must accumulate all command's output in one buffer and ignore any strange output. Because a command can take long time to complete, sending a command is an asynchronous function call.

As mentioned above, *Command Shell Proxy* works with pipes but doesn't open or close these pipes. *Qemu Proxy* is responsible for providing and exporting them. In Figure 2 *Command Shell Proxy* sends a command which eventually gets written to standard input of a Qemu process and thus to a guest OS shell. And vice versa, command output is forwarded by Qemu to its standard output and it gets received by *Command Shell Proxy* via pipe provided by *Qemu Proxy*.

To summarize, *Command Shell Proxy* must:

- write a user supplied command to guest OS;
- accumulate all command output;
- ignore any output with no currently running command;
- allow for only one command to be executed at a time;
- notify user about command completion.

Task.

Running one command at a time in an automated manner on a particular VM is already a considerable achievement. But nearly all experiments involve several nodes and require the execution of several programs on them simultaneously. In addition to the strict need to run multiple commands on nodes simultaneously, there is also a possibility to execute several commands in parallel just to speed up the overall experiment. For example, initial guest OS configurations can be performed in parallel though there is usually no special requirement for this. Let's name a group of commands that must start their execution simultaneously on different VMs as a *Task*. A *Task* completes with completion of the last command in it. No *Task* can start its execution until previous *Task* is completed. The end of a *Task* can be interpreted as an implicit barrier for all commands that belong to the same *Task*. Several *Task*s should not be started in parallel.

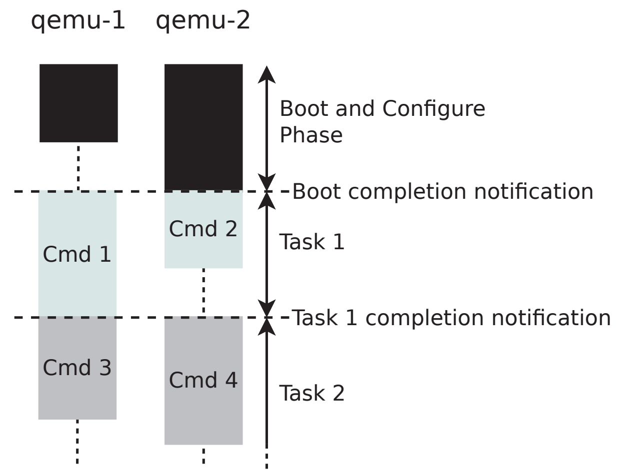

Figure 3: Command synchronization with *Task*s

Usually *Task*s start their execution just after boot and initial setup of all guest OSes, so there is an easy way to configure *Topology Helper* to start the first *Task* immediately after initial configuration. *Task*s execute in right order because every *Task* knows which *Task* to execute next. A *Task* with no successor is the last *Task* to be executed. User scenario can analyze commands' output and dynamically add *Task*s to the execution queue. An example simulation run with two "qemu" nodes and two *Task*s is shown in Figure 3 to illustrate synchronizing nature of *Task*s.

The *Task* must:

- start simultaneous execution of own commands upon start;
- stop upon stop of all own commands;
- analyze commands output (should be implemented in subclasses of *Task*);
- start next *Task* in chain.

4. QEMUNET PERFORMANCE

Qemunet approach is to run multiple Qemu virtual machine instances on a single host machine to emulate nodes behavior. Qemu provides full virtualization VM. Full virtualization is known to have the most performance penalties among other virtualization solutions. So scalability of the entire Qemunet system is questionable. In this section we present results of basic performance evaluation experiments.

A measure of performance for a simulation or emulation system is a number of events that this system is capable to dispatch without any significant distortion of results. For a network simulator a rate of events is strongly correlated with a rate of packets being processes. Thus our performance-biased experiments are aimed at estimating a total number of packets per second that Qemunet system is able to sustain (system throughput).

In addition, virtualized experiments make sense only for the number of nodes exceeding the number of real stations that someone can physically manage. This number varies

Figure 4: Topology is a chain of nodes

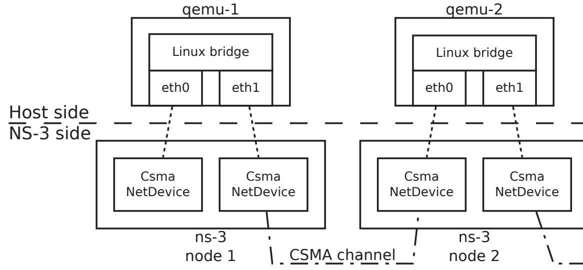

Figure 5: Connecting two nodes in a chain

from experiment to experiment and from type to type of real stations. But generally someone could hardly manage more than 10 real nodes. So we are going to show that Qemunet system is able to instantiate more than 10 nodes even under heavy load.

Performance evaluation experiments were built around `iperf` client and server sending UDP traffic across a chain of nodes from the first to the last one (see Figure 4). The topology was chosen to be a chain of nodes in order to ensure that every packet goes through every node (thus simulating the worst-case scenario). Such topology choice allows for simple conversion between end-to-end throughput and whole system performance expressed in packets per second.

More details on nodes and links configurations are shown in Figure 5 and described below. Nodes are connected with ns-3 CSMA links configured with unlimited data rate and no additional delay and packet loss. The model of CSMA channel is lightweight enough to measure nearly pure simulator overhead. All nodes are of "qemu" type meaning that a Qemu VM (with Linux running inside it) is created for each node to define node's behavior. It is possible to create fully simulated ns-3 nodes, but that will only increase ns-3 simulator pressure and decrease end-to-end throughput. Each node has two CSMA interfaces connected to two corresponding adjacent nodes. Thus two Gigabit Ethernet devices are emulated for guest OS by Qemu (actually, small overhead VirtIONet devices are used). Nodes are configured to bridge these two devices. As all nodes are "qemu" nodes, the bridging is implemented via guest Linux kernel bridge. All packet checksumming and PCAP trace files writing are disabled to greatly improve system performance.

Ns-3 realtime scheduler is left in the default best effort mode. Measuring performance in hard limit mode would be more fair but overloading simulator in this mode makes the whole system quite fragile hampering the experiment.

At the considered ns-3 revision, ns-3 TapBridge NetDevice does not perform any receive queue management. At packet rates exceeding ns-3 throughput the receive queue starts unlimited growth. Sooner or later this will lead to memory exhaustion and simulation termination. So we have chosen test duration not to be very long and thus not to crash even

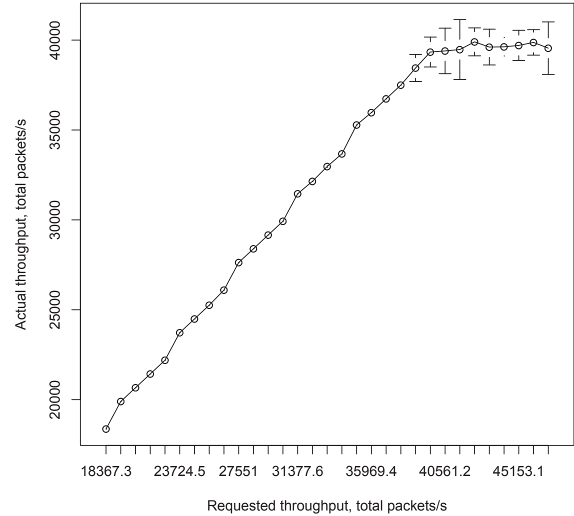

Figure 6: Measured throughput as the function of requested throughput in the chain of 10 qemu nodes. 95% confidence intervals are shown

under heavy load. A similar problem was described and already solved for EmuNetDevice [22].

A server, hosting our virtualized testbed, which was used to evaluate performance, is approximately two times processor count and RAM rich compared to a modern desktop computer (Intel Xeon E5520 2.27GHz with HTT and Intel VT-x, 8 real cores, 12 GB of RAM). Intel VT-x and Kernel-based Virtual Machine (KVM) were used by Qemu to reduce virtualization overhead.

In all experiments the `iperf` client is trying to send some requested number of packets per second to the `iperf` server while server measures actual number of packets delivered per second. Both numbers are multiplied by $N-1$ in the chain of N nodes to obtain total requested and measured network throughput. UDP packet payload size is fixed to `iperf`'s default 1470 bytes
$(\underbrace{\text{payload}}_{1470} \approx \underbrace{\text{Ethernet payload}}_{1500} - \underbrace{\text{IP and UDP headers}}_{20+8})$.

Measured network throughput as the function of requested network throughput in the chain of 10 nodes is presented on Figure 6. It is clearly shown that in this case Qemunet performance is saturated at some critical requested throughput near $4 \cdot 10^4$ packets/s. An obvious bottleneck is ns-3 simulator core which is unable to process more packets per second. Ns-3 peak CPU utilization was about 150% loading 1.5 cores in average with main simulator thread and Tap-device reader threads. Note, that we used a relatively simple CSMA channel model instead of much more complex WiFi model in these experiments.

The same saturation behavior was observed for chain length ranging from 2 to 32 nodes, in all cases critical throughput was less than $4 \cdot 10^4$ packets/s. Therefore, we have defined critical network throughput as the value of measured

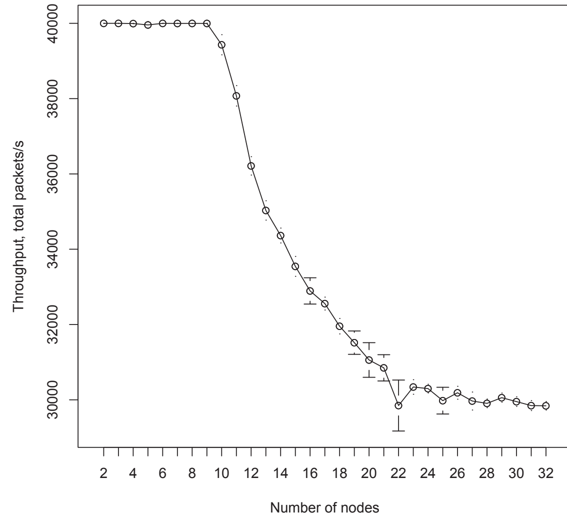

Figure 7: Measured throughput as the function of number of nodes in the chain. Requested throughput is $4{\cdot}10^4$ packets/s. 95% confidence intervals are shown

throughput for requested throughput of $4{\cdot}10^4$ packets/s. Measured this way critical network throughput as the function of chain length is shown on Figure 7. The critical throughput does not depend on the number of nodes up to 9-10 nodes and then degrades from $4{\cdot}10^4$ packets/s to $3{\cdot}10^4$ packets/s when number of nodes grows from 10 to 32. We attribute this decrease to the increasing CPU pressure from Qemu processes to the main ns-3 simulator thread.

From these basic results we conclude that Qemunet performance is sufficient to run $O(10)$ nodes with the maximal network load of $O(10^4)$ packets/s. Ns-3 core is a performance bottleneck in this region and much larger number of nodes may be emulated when traffic load is small. Overall system performance can be increased by employing more efficient packet exchange mechanism between ns-3 and Qemu processes.

5. CONCLUSION

In this work we have described Qemunet system targeted to facilitate, automate and introduce better control into running real-time simulations with multiple VMs involved. Qemunet system is built upon thought-out architecture that allows extending it to support different VMs, topology description formats and etc. We have shown some performance benchmarks of the whole system to provide some basis for estimating possible applications of the proposed system. The source code of Qemunet is not yet publicly available but we're working on getting it to public.

Interestingly, Qemunet system can be used to perform automated unit, system and regression testing of software being under development. For example, a *Task* can be easily wrapped into a *Test Case* providing a nearly complete framework for testing.

Future work may be done to make Qemunet more configurable, flexible and stable; to provide interoperability with other closely related projects.

6. REFERENCES

[1] QEMU, a generic machine emulator and virtualizer. `http://wiki.qemu.org/`.

[2] VMware, a provider of virtualization software. `http://www.vmware.com/`.

[3] VirtualBox, a powerful x86 and AMD64/Intel64 virtualization product. `http://www.virtualbox.org/`.

[4] Xen, a virtual-machine monitor. `http://www.xen.org/`.

[5] User-mode Linux (UML), enable multiple virtual Linux systems to run as an application. `http://user-mode-linux.sourceforge.net/`.

[6] OpenVZ, a container-based virtualization for Linux. `http://openvz.org/`.

[7] Linux Containers, virtualization system for computers running GNU/Linux. `http://lxc.sourceforge.net/`.

[8] FreeBSD Jail, implementation of OS-level virtualization. `http://www.freebsd.org/doc/en_US.ISO8859-1/books/arch-handbook/jail.html`.

[9] Network Simulator 3. `http://www.nsnam.org`.

[10] Common Open Research Emulator (CORE). `http://cs.itd.nrl.navy.mil/work/core/index.php`.

[11] J. Ahrenholz, C. Danilov, T.R. Henderson, and J.H. Kim. CORE: A real-time network emulator. In *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pages 1–7, 2008.

[12] Mathieu Lacage. *Experimentation Tools for Networking Research*. PhD thesis, 2010. Universite de Nice Sophia-Antipolis, Supervisor-Dabbous, Walid.

[13] Mathieu Lacage, Martin Ferrari, Mads Hansen, Thierry Turletti, and Walid Dabbous. Nepi: using independent simulators, emulators, and testbeds for easy experimentation. *SIGOPS Oper. Syst. Rev.*, 43:60–65, January 2010.

[14] Virtual Distributed Ethernet (VDE). `http://vde.sourceforge.net/`.

[15] libpcap, a portable C/C++ library for network traffic capture. `http://www.tcpdump.org/`.

[16] ns-3 Realtime Scheduler. `http://www.nsnam.org/wiki/index.php/Emulation_and_Realtime_Scheduler`.

[17] The ns-3 topology generator. `http://www.nsnam.org/wiki/index.php?title=Ns3Generator`.

[18] Python subprocess module, subprocess management. `http://docs.python.org/library/subprocess.html`.

[19] IPC::Run, allows you to run and interact with child processes using files, pipes, and pseudo-ttys. `http://search.cpan.org/perldoc?IPC::Run`.

[20] Expect, a tool for automating interactive applications. `http://www.nist.gov/mel/msid/expect.cfm`.

[21] ns-3 Bug 631: RealtimeSimulatorImpl not compatible with python bindings. `http://www.nsnam.org/bugzilla/show_bug.cgi?id=631`.

[22] ns-3 Bug 939: EmuNetDevice uses too much memory when reading packet bursts. `http://www.nsnam.org/bugzilla/show_bug.cgi?id=939`.