NCTUns Distributed Network Emulator

Shie-Yuan Wang¹ and Yu-Ming Huang
Department of Computer Science
National Chiao Tung University, Hsinchu, Taiwan

Abstract

Emulation enables real-world devices to interact with a network simulated in real time. It is a very useful approach to testing the functions and performances of real-world devices under various simulated network conditions. To conduct a network emulation, the clock of the network emulator needs to be accurately synchronized with the real-world clock. However, when the load of the network emulator is too high, which slows down the advances of its clock, accurately synchronizing its clock with the real-world clock becomes impossible.

To overcome this problem, in this paper we propose a distributed network emulation approach. This approach partitions a large network into several smaller parts and lets each part be emulated by a different machine. We have applied it to the NCTUns network simulator and emulator and developed a highly-integrated NCTUns distributed emulation environment. On this environment, one can easily set up, conduct, and control a multi-machine distributed emulation on a single NCTUns machine. In this paper, we present the design of this approach, introduce the operations of the NCTUns distributed emulation environment, and evaluate its emulation performances.

1. Introduction

Network emulation is an approach that enables real-world devices to interact with a network simulated in real time. Most existing network emulators such as NIST Net [1] abstract a complex network as a router with various packet treatments. They use a machine to run such a router in real time and connect real-world devices to it. The packets generated by the real-world devices are directed to the router. When these packets pass through the router, the router gives them special treatments such as dropping, delaying, or reordering to simulate the behavior of the complex network that is under study. By this approach, one can evaluate the functions and performances of a real-world device under various simulated network conditions.

The NCTUns network simulator and emulator [2] is an innovative tool that seamlessly integrates simulation and emulation to provide unique advantages over most existing network emulators. In a NCTUns emulation, the simulated network that real-world packets will pass through need not be abstracted as a single router as NIST Net does. Instead, the simulated network can be a large network composed of a large number of nodes and links, each with its own detailed attribute settings. Actually, this simulated network, although used in an emulation,

¹ E-mail address: shieryuan@csie.nctu.edu.tw
can be as complex as any network that can be simulated by NCTUns. NCTUns uses a novel kernel reentering simulation methodology [3], [4], [5], [6] so that simulated nodes can readily run real-world applications and use the real-world Linux TCP/IP protocol stack to exchange packets. Due to this property, in a NCTUns simulation, realistic traffic flows generated by real-world applications and TCP/IP protocol stack can be set up among nodes in the simulated network. When such a simulated network is purposely run in real time (for brevity, we will simply call such a simulated network the “emulated network” in the rest of the paper), real-world devices can exchange their packets over the emulated network. These real-world packets can encounter the background traffic (e.g., HTTP/TCP traffic) generated inside the emulated network to experience more realistic network conditions caused by these traffic.

In addition to allowing two real-world devices to exchange their packets over an emulated network (this usage is commonly supported by traditional network emulators), NCTUns allows the application running on a real-world device to set up real TCP/UDP connections with a real-world application running on a (simulated) node in the emulated network. That is, for a real-world device whose application is to be tested, if its application can be run on the NCTUns machine (which runs on the Linux kernel), one can create a (simulated) node in the emulated network and run up that application on the created node to represent that real-world device. The (real-world) application running on such a simulated node can interact via real TCP/UDP connections with the applications running on any real-world device. By changing the behavior of the application running on a simulated node, one can observe whether the application running on a real-world device would respond correctly to the various (normal or abnormal) requests issued by that application.

With this capability, if a real-world device under emulation study uses the Linux kernel as the platform for its applications, one can run its applications on the single NCTUns machine without using and connecting the device to the NCTUns machine. On the other hand, if the device uses an operating platform other than Linux for its applications (e.g., Windows), then one still needs to connect the device to the NCTUns machine for an emulation study unless the source code of the application is available and the application can be rebuilt on the Linux platform. This capability can save the number of real-world devices that need to be used to run up applications for an emulation study.

The above single-emulation-machine approach works well in most conditions. However, it may encounter scalability problems with a very large emulation case. For example, when (1) the number of real-world devices that need to connect to a NCTUns emulated network is large, (2) the size (i.e., the number of nodes and links) of the emulated network is large, (3) the number of applications that need to run up on the nodes in the emulated network is large, (4) these applications consume a large amount of main memory and CPU computing cycles during their executions, or (5) the amount of packets generated by these applications and exchanged over the emulated network is large, this approach may not be scalable. In the following, we elaborate on these problems.

The first problem is that the NCTUns emulation machine may not have enough network interface cards (NIC) to independently connect to each of these devices. Ideally, the NCTUns emulation machine should use a different NIC to connect to a different device so that these devices’ traffic into or out of the NCTUns machine will not be affected by one another. However, nowadays a PC normally can only accommodate four NICs at the most. The second problem is that the emulation machine may not be fast enough to run the emulated network in real time. For example, when a heavy aggregate packet traffic load is generated by these
devices and these packets need to pass through the complex emulated network, the NCTUns emulation machine may fail to simulate the processings of these packets in the emulated network in real time. In such a case, the emulation fails to be an emulation. The third problem is that, under a heavy load condition, even though the NCTUns emulation machine can barely run the emulated network in real time the synchronization between its simulation clock and the real-world clock will become less accurate. This may cause the real-world packets exchanged over the emulated network to experience unnecessary extra delays or losses. The fourth problem is that, when many applications need to be run up on the emulated network, the NCTUns emulation machine may not have enough main memory space to accommodate all of these applications at the same time.

To overcome these problems, in this paper we propose a distributed emulation approach for NCTUns. By this approach, a large complex emulated network is divided into several smaller parts and each part is run by a NCTUns emulator on a different machine. Each NCTUns emulator just needs to (1) emulate the nodes and links in its assigned part, (2) run the real-world applications that should be run up on the nodes in its assigned part, (3) exchange real-world packets between it and the real-world devices that are attached to some nodes in its assigned part, and (4) exchange packets with other NCTUns emulators (running on other machines) when these packets need to traverse into other parts. By properly dividing a large complex emulated network into several smaller parts such that each part can be emulated by NCTUns in real time, NCTUns can emulate a very large complex network with many real-world application programs launched on it and many real-world devices attached to it. The scale of such a distributed emulation is only limited by the number of available machines that can participate in the distributed emulation. The rest of the paper is organized as follows. In Section 2, we survey related emulators. In Section 3, we present the original design and implementation of the NCTUns network emulator. In Section 4, we present a new design and implementation for the NCTUns distributed emulation approach. In Section 6, we present the performance and scalability results of the proposed approach. Finally, we conclude the paper in Section 7.

2. Related Work

NS2 [7] is mostly used as a network simulator but in its user manual it says that it also supports emulation function. According to its user manual, it can intercept and direct realworld packets to its simulated network and use its real-time scheduler to synchronize its simulation clock with the real-world clock. NCTUns, on the other hand, uses an emulation kernel module to intercept and direct real-world packets into its simulated network and uses periodic events to perform time synchronization. These two network simulators can intercept and direct real-world packets into a complex simulated network running in real time. In contrast, NIST Net is purely a network emulator and not a network simulator. As a result, the real-world packets intercepted by itself cannot be directed to a complex simulated network running in real time. Instead, they can only receive simple treatments such as delaying, reordering, or dropping when they pass through the machine running the NIST Net kernel module. Currently, NS2 and NIST Net do not support distributed emulations. They can only use a single machine to run an emulation.
Emulab [8] is a network testbed that provides researchers with an environment in which to develop, debug, and evaluate their systems. Emulab refers both to a facility and to a software system. An Emulab installation may be composed of a handful of nodes (PC) or up to hundreds of nodes. Through a user interface and a software system, researchers can specify and conduct an experiment on an Emulab to test the functions and performances of network protocols or applications. In an Emulab, each node is a real machine (not a virtualized one) and the user can fully control it with the “root” accesses. The primary difference between Emulab and NCTUns is that in Emulab each physical node (e.g., a PC) can only emulate a node of an experiment. Therefore, the number of nodes of an experiment that can be conducted on an Emulab must be less than the number of physical nodes in an Emulab. In contrast, because in NCTUns a physical node can emulate multiple (e.g., hundreds of) nodes of an experiment, such a restriction does not exist.

CORE [9] (Common Open Research Emulator) is a tool that allows researchers to emulate entire networks on one or more machines. CORE consists of a GUI for easily drawing topologies that drives virtual machines, and various utilities. CORE uses virtualized network stacks in a patched FreeBSD kernel or Linux virtual machines. Like NCTUns, a machine on CORE can emulate multiple nodes of an experiment. However, since in CORE a virtual machine is used to emulate a node of an experiment and a virtual machine needs to consume much CPU cycles, memory space, and disk space resources, a machine in CORE can only emulate a handful of nodes of an experiment. In contrast, since NCTUns uses the lightweight kernel-reentering simulation methodology, which allows it to emulate hundreds of nodes on a machine, the number of nodes that can be emulated in NCTUns is much larger than that in CORE.

In [10], the authors extended the emulation capability of NCTUns to support mobile networks. In the original emulation design of NCTUns, it allows a real-world mobile node to exchange packets with the emulated network. However, such a node must be a terminal node (e.g., a WLAN client device) and cannot be a gateway to another network that can be mobile (e.g., an Ethernet fixed network on an airplane). To provide such an extension, in a network topology the authors chose a host or router as a gateway to divide the network into two parts and let each part be emulated by a different machine. These two machines may exchange packets if packets need to pass through the gateway during emulation.

The approach proposed in [10] can be viewed as a distributed simulation approach similar to what we propose in this paper. However, there are several important differences between the two approaches and in the following we elaborate on one of them. The approach proposed in [10] is an ad-hoc approach that needs tedious and careful manual modifications to many network configuration files (e.g., the routing table file, IP filtering rules file, etc.) generated by NCTUns. This will become a serious problem when the number of gateways used in the emulated network is large as the degree of modifications that need to be made to the original files will grow greatly.

In contrast, with intelligence, our approach is a fully automatic approach that needs no manual modifications to these files. To perform a NCTUns distributed emulation, one first decides how to divide the whole emulated network into several parts. Then one uses the “virtual router” icon, which represents a gateway between two parts, to connect any two parts together where appropriate. After these simple GUI operations, the GUI console will intelligently and automatically generate appropriate network configuration files and send them to all machines participating in the distributed emulation. When the distributed
emulation is done, the GUI console will collect emulation results and packet log files from all participating machines and intelligently merge these files together as if the emulation were totally performed on just one machine. Using the NCTUns distributed emulation approach, the user need not worry about how to modify the contents of the generated network configuration files to correctly perform a distributed emulation. Instead, the user can enjoy the many benefits of distributed emulation without such worries and headaches. In Section IV, the detailed design and implementation of our approach will be presented.


For a reader to understand the design and implementation of NCTUns distributed emulation, it is necessary for him/her to first understand the design and implementation of NCTUns single-machine emulation. Therefore, in the following, we first present the design and implementation of single-machine emulations.

3.1. The Architecture

NCTUns is both a network simulator and a network emulator. It has seven main components: GUI, job dispatcher, coordinator, simulation engine, patched Linux kernel, applications, and daemons. The GUI program enables a user to create, set up, and control a simulation/emulation execution easily. The coordinator program is run on every participating machine to monitor its current busy/idle status and report the status to the job dispatcher program. The job dispatcher program can monitor the statuses of multiple participating machines by receiving the status reports sent from the coordinator programs running on these machines. Fig. 1 shows this distributed simulation architecture. This architecture was originally designed to support concurrent simulations to increase simulation throughputs. Later on, we found that it works equally well for distributed emulations and thus directly used it to support distributed emulations.

Figure 1. The architecture of NCTUns for distributed emulations.
After a user creates a simulation/emulation case, the GUI sends a request to the job dispatcher to ask for an idle coordinator and then sends the simulation job to the chosen coordinator. After the coordinator receives the job, it forks a simulation engine process to run the simulation case. During simulation, when an application running on a node sends a packet to another application running on a different node, the packet will pass through the standard socket interface, the Linux TCP/IP protocol stack, and arrive at a special virtual interface called the "tunnel interface" in the kernel. The simulation engine will retrieve the packet from the tunnel interface and use its protocol modules to simulate the MAC/PHY operations and effects for the packet. After these processings, the simulation engine will write the packet into another tunnel interface in the kernel. If that tunnel interface is the interface used by the destination node of the packet, the packet will pass through the Linux TCP/IP protocol stack and socket interface and finally be received by the application running on the destination node. Fig. 2 gives an overview of the relationships among applications, simulation engine, kernel, tunnel interfaces, etc. Due to space limitation, this paper can only briefly describe the NCTUns architecture. More information about its simulation methodology and architecture is available in [3], [4], [5], [6].

3.2. The S.S.D.D IP Address Scheme

NCTUns has two unique and useful features: (1) Any real-world application programs can directly run on simulated nodes to exchange packets (2) The real-world Linux TCP/IP protocol stack is directly used to generate realistic simulation results. To provide these
advantages, a special IP address scheme, which we call the S.S.D.D IP address scheme, is internally used in NCTUns. In this scheme, each network interface (i.e., tunnel interface) in a simulated network is assigned an IP address of the 1.0.SubnetID.HostID format. Suppose that node A’s IP address is 1.0.SubnetA.HostA and node B’s IP address is 1.0.SubnetB.HostB. If node A wants to send a packet to node B, node A should use SubnetA.HostA.SubnetB.HostB as the destination IP address for that packet. Conversely, if node B wants to send a packet to node A, node B should use SubnetB.HostB.SubnetA.HostA as the destination IP address for that packet. From this rule, it is clear that if a source node wants to send a packet to a destination node, the destination IP address used for that packet is formed by concatenating SubnetID, HostID of the source node with those of the destination node. This is why we call this scheme the S.S.D.D scheme.

This scheme is used to avoid a route conflict problem in the system routing table. In the real world, each node has its own routing table to store its own routing entries. However, because NCTUns uses the system routing table in the kernel to store all (simulated) nodes’ routing entries, these entries may conflict with each other. Using Fig. 3 as an example. Suppose that the S.S.D.D scheme is not used. Then the host on the left will have a routing entry of (1.0.2.1, tunnel 1) and the router in the middle will have a routing entry of (1.0.2.1, tunnel 4), where the first object in the entry represents an IP address while the second object indicates which interface should be used to forward out a packet with this IP address as its destination address. Because NCTUns uses the kernel’s routing table to store these two entries, the kernel will get confused as to which interface should be used to forward such a packet. On the other hand, if the S.S.D.D scheme is used, the (1.0.2.1, tunnel 1) entry on host 1 becomes (1.1.2.1, tunnel 1) and the (1.0.2.1, tunnel 4) entry becomes (2.2.2.1, tunnel 4). With these changes, when these two entries are stored in the kernel’s routing table, no routing conflicts will result.

For a packet, this S.S.D.D scheme is not only applied to it at the source node but also at all forwarding nodes along the path to its destination node. Suppose that there is a forwarding
node with 1.0.SubnetF.HostF on the path of the packet whose destination node is node B. Then before the packet enters this forwarding node, the simulation engine will have changed the destination IP address in the IP header of the packet to SubnetF.HostF.SubnetB.HostB. This operation can be seen in Fig. 3 when the packet is about to enter tunnel 2 and 3. With this design, when a packet arrives at its destination node, the S.S.D.D destination IP address in its IP header will have the property that the S.S is equal to the D.D. This property is used as a rule to determine whether a packet has arrived at its destination node or should be continuously forwarded.

Another design is to apply the above S.S.D.D operations to the source address of a packet as well. This operation can be seen in Fig. 3. The purpose of doing this operation is that after a packet has arrived at its destination node, the destination node may need to send back a reply packet (e.g., a TCP ACK packet) to the source node of the packet. With this design, the destination node can immediately take and use the S.S.D.D source address of the packet as the S.S.D.D destination address for the reply packet. For example, in Fig. 3, when the packet arrives at the host on the right, its S.S.D.D source address is 2.1.1.1, which is exactly the same as the S.S.D.D destination address that should be used by the host on the right if it wants to send a packet to the host on the left.

The S.S.D.D IP address scheme is the core scheme of NCTUns. It is internally used in NCTUns and invisible to applications and users. We have modified the Linux kernel so that in an NCTUns simulation case, an application can still use the normal 1.0.SubnetID.HostID format to specify its destination IP addresses.

3.3. The S.S.D.D IP Address Scheme

In the topology editor of NCTUns GUI, each real-world device that participates in emulation is represented by a node icon and is called an "external host" or "external router," depending on whether it is a host or router in the real world. In the topology editor, the icon of this "external node" (say, node A) can be connected to the icon of any (simulated) node (say, node B) in the emulated network by an emulated link (say, link C) to indicate how the real-world device is connected to the emulated network. With this specification, when the packets generated by the real-world device enters the NCTUns emulation machine through a physical link, these packets appear to be generated from node A and will enter the emulated network via link C. For these packets, node B will be the first (simulated) device that they encounter after they enter the NCTUns emulation machine. On the other hand, if there are packets to be sent to the real-world device from the NCTUns emulation machine, regardless of whether these packets are generated by some (simulated) nodes in the emulated network or they are generated by other real-world devices, these packets will traverse and leave the emulated network via node B and link C to reach node A, which represents the real-world device to receive these packets. Using Fig. 4 as an illustration, node A corresponds to node 4, node B corresponds to node 3, and link C corresponds to the link connecting node 4 and node 3 in the GUI topology editor. The network composing of node 1, node 2, and node 3 and the three links is emulated (simulated in real time) by the emulation machine. The emulation machine is physically connected with the external host via a highbandwidth short-delay link such as a 100 Mbps Ethernet crossover cable.
In NCTUns, suppose that an application P running on a node in the emulated network wants to exchange packets with an application Q running on a real-world device. In this case, the destination IP address used by application P for such packet exchanges should be the IP address of node A in the emulated network rather than the IP address used by the real-world device. This design is adopted as these packets need to be routed correctly within the emulated network to reach node A before the NCTUns emulation machine sends them to the real-world device. Using Fig. 4 as an illustration, suppose that application P is running on node 1 and application Q is running on the external host in the real world, then the destination IP address used by application P should be 1.0.2.1 rather than the IP address used by the external host in the real world (which, for example, may be 192.168.10.77).

The NCTUns emulation machine uses a kernel module to capture the packets to be sent to a real-world device. Using the above notation, the capture is done when packets have traversed the emulated network and reached node A. When an emulation is started, the kernel module is invoked and then it reads the rules specified in a configuration file. These rules associate the IP address used by a real-world device with the IP address (of the 1.0.SubnetID.HostID format) assigned to its corresponding node (i.e., node A) in the emulated network. The kernel module uses the Linux netfilter mechanism to capture all packets flowing in the kernel. It checks whether their destination IP addresses match the IP address of node A. For a captured packet, if such a match is found, it indicates that this packet has traversed the emulated network and reached node A and now it is time to send the packet to the corresponding real-world device. Before sending the packet, the kernel module changes the destination IP address in the packet header to the IP address of the corresponding real-world device. Such an address translation is necessary as doing so allows the packet to be successfully received by the real-world device. From this perspective, the NCTUns emulation machine acts like a NAT (Network Address Translator) when performing an emulation.

3.4. Single-Machine Emulation with External Router

An external router can also interact with an emulated network. This is a very useful feature as traffic originated from the emulated network can be directed to the router,
experience the router’s packet processings, and then return back to the emulated network. With this capability, one can easily test the router’s functionality (e.g., sending virus and host-attacking packets to see whether the router can detect them).

Fig. 5 shows an example where three emulated hosts are connected to an external router (the middle one). On this topology, we set up two greedy TCP connections. The first one starts at host 1 and ends at host 3 while the second one starts at host 2 and ends at host 3. The packets of these two TCP connections need to pass through the external router. That is, they need to leave the emulated network (after leaving host 1 and host 2), enter the real-world network to reach the external router, leave the router, and then reenter into the emulated network (to reach host 3).

The physical network setup for running this emulation is shown in Fig. 5. The emulation machine needs to have three network interfaces each of which connects to one port of the external router. Clearly, there are three physical links needed to connect these network interfaces to these ports — one to one. These physical links are represented as links in the emulated network respectively, and the bandwidths and delays of these links are specified in and emulated by the emulation machine. As in the external host case, once real-world packets enter the emulated network, they need to traverse emulated links and are subject to the bandwidths and delays of these emulated links.

The emulation machine needs some information about the external router. For each port of the external router in the real world, the user needs to provide the association among the following information entities to the emulation machine: its port ID, its assigned IP address in the emulated network (this information is automatically provided by the GUI), the real IP address of the network interface on the emulation machine that connects to this port via a link,
the name (e.g., eth1) of the above interface (on the emulation machine), and the real IP address used by this port on the external router. Using the emulation network in Fig. 5 as an example, assuming that the names of the interfaces with 140.113.1.1, 140.113.2.1, and 140.113.3.1 on the emulation machine are eth1, eth2, and eth3, respectively, then the association table should contain the following entries: (1, 1.0.1.2, 140.113.1.1, eth1, 140.113.1.2) for port 1, (2, 1.0.2.2, 140.113.2.1, eth2, 140.113.2.2) for port 2, and (3, 1.0.3.2, 140.113.3.1, eth3, 140.113.3.2) for port 3.

On the external router in the real world, some routing entries need to be added to its routing table so that packets originated from the emulated network can be redirected back to the emulated network. The rules for generating these routing entries are as follows. For every host with 1.0.X.Y as its assigned IP address in the emulated network, we need to use the following commands to add the needed routing entries: “route add 200.Z.X.Y dev NICNAME” or “route add 200.Z.X.Y gw GatewayIPaddress.” Here Z is a variable taken from the set of all subnet IDs used in the emulated network, NICNAME is the name of the interface on the external router (e.g., fxp0 or eth0), and GatewayIPaddress is the IP address of the interface on the emulation machine to which the external router would like to send packets with 200.Z.X.Y as their destination IP address. Note that “200” must be used because the emulation kernel module purposely changes the destination IP address of a packet (e.g., 1.0.X.Y) going to the external router to 200.0.X.Y, so that this type of packets can be recognized by the external router and later be sent back by the external router to the emulated network.

Beside changing the first number of the destination IP address from the default “1” to “200,” the emulation kernel module also changes the second number of the destination IP address from the default “0” to the ID of the subnet where the packet leaves the emulated network to enter the external router. This design is required. Without such a design, if there are two external routers in an emulated network, when a packet enters the emulated network after being sent to an external router and then coming back, the emulation kernel module will not be able to know from which subnet this packet should continue its journey in the emulated network. For this reason, assume that host 2 on the left sends a packet to host 3 on the right. Then the destination IP address of the packet will be changed to 200.2.3.1 before being sent to the external router.

Using the above example to illustrate, suppose that link 1 is subnet 1, link 2 is subnet 2, and link 3 is subnet 3 in the emulated network. Further suppose that the IP address of host 1 is 1.0.1.1, the IP address of host 2 is 1.0.2.1, and the IP address of host 3 is 1.0.3.1, and the IP address of the external router on link 1 is 1.0.1.2, the IP address of the external router on link 2 is 1.0.2.2, and the IP address of the external router on link 3 is 1.0.3.2. Suppose that in the real world the real IP address used by the external router port configured with 1.0.1.2 in the emulated network is 140.113.1.2, the real IP address used by the external router port configured with 1.0.2.2 in the emulated network is 140.113.2.2, and the real IP address used by the external router port configured with 1.0.3.2 in the emulated network is 140.113.3.2. Further suppose that on the emulation machine the IP address of the interface connecting to link 1 is 140.113.1.1 in the real world, the interface connecting to link 2 is 140.113.2.1, and the interface connecting to link 3 is 140.113.3.1. These address settings are best understood by viewing Fig. 5.

For this example, on the external router we need to execute the following route commands to add the required entries:
These commands have the effect that all packets that originate from subnet 1, 2, or 3 and go to 1.0.1.1 will be sent to 140.113.1.1 via link1. Similarly, these commands have the effects that all packets that originate from subnet 1, 2, or 3 and go to 1.0.2.1 will be sent to 140.113.2.1 via link 2 and all 7 packets that originate from subnet 1, 2, or 3 and go to 1.0.3.1 will be sent to 140.113.3.1 via link 3. If there are multiple hosts on a subnet in the emulated network, it is more efficient and convenient to use subnet routing rather than host routing to specify these routing entries. The following shows the route commands for adding these subnet routing entries:

```
route add -net 200.1.1.0/24 gw 140.113.1.1
route add -net 200.2.1.0/24 gw 140.113.1.1
route add -net 200.3.1.0/24 gw 140.113.1.1
route add -net 200.1.2.0/24 gw 140.113.2.1
route add -net 200.2.2.0/24 gw 140.113.2.1
route add -net 200.3.2.0/24 gw 140.113.2.1
route add -net 200.1.3.0/24 gw 140.113.3.1
route add -net 200.2.3.0/24 gw 140.113.3.1
route add -net 200.3.3.0/24 gw 140.113.3.1
```

4. Design and Implementation for Distributed Emulations

NCTUns employs a central controller to manage all emulation machines participating in a distributed emulation. The functions of the central controller are mainly implemented in the GUI. The architecture of NCTUns for distributed emulations is shown in Fig. 1. It is the same as the architecture of NCTUns used to support distributed concurrent simulations.

For a distributed emulation, one uses the GUI to divide the network topology of a case into serveral parts. Each part is an area of the emulated network and can include several external hosts. After a user specifies and configures these parts, the GUI sends the information about these parts to participating machines. After a participating machine receives the information about the part assigned to it, it runs up a NCTUns simulation engine process to emulate the assigned part. When the emulation of the assigned part is finished, the results and packet logs generated during the emulation will be sent back to the GUI for integration. Finally, when the emulations of all parts are finished and their results and logs have been sent back to the GUI, the GUI integrates all these results and logs together and the
distributed emulation across multiple machines is finished. In the following, we elaborate on these operations.

4.1. Topology Division

In a distributed emulation, partitioning the topology is the first step. Because the GUI lacks the intelligence to automatically partition a topology in a load-balancing fashion, this step is done by a user. In the GUI, a new node called "virtual router" is used to act as a boundary node between two parts. Fig. 6 shows a topology which is composed of 4 hosts, 2 switches, and one virtual router (on the middle). The two hosts and one switch on the left constitute part 1 while the two hosts and one switch on the right constitute part 2. The virtual router divides the whole topology into two parts and connects them together. If a packet is sent from node 1 to node 3, it will need to pass through the virtual router. This means that in the real world the packet will be sent by the machine emulating part 1 to the machine emulating part 2 via an entity that represents the virtual router.

![Figure 6. A topology is divided into two parts that are connected together by a virtual router.](image-url)

4.2. Virtual Router

In the real world, a virtual router can represent a real router or simply a crossover cable (or a layer-2 switch, which can function as a crossover cable). Fig. 7 shows the configuration dialog box for a virtual router, which is composed of two panels that are exclusive to each other.

If the virtual router is used to represent a real router, one should tick the "Use a real router to connect multiple emulation machines" option to enable the top panel. On this panel, one can specify which machine is used to emulate a part of the network. For each port of a virtual router in the emulated network, it connects to a part and has a corresponding entry in the table on the first panel. Each entry has five fields: 1) Port ID; 2) Assigned IP address (in the emulated network); 3) Coordinator IP address; 4) Emulation machine IP address; and 5) the IP address of the network interface on the real router that corresponds to this port. The meanings of these fields are explained below.
The “Port ID” field denotes the ID of the port. The “Assigned IP address” field is the assigned IP address for this port in the emulated network. The values of these two fields are automatically assigned and provided by the GUI.

The “Coordinator IP address” field specifies the IP address of the emulation machine whose coordinator program is responsible for forking a simulation engine process to emulate the part that this port connects with. If a part is connected to multiple virtual routers, then for the ports of these virtual routers that connect to this part, their “Coordinator IP address” fields should be set to the same, and it should be the IP address of the emulation machine that emulates this part. In NCTUns, if an emulation machine has multiple network interfaces (and thus has multiple IP addresses), its coordinator will choose one to register with the dispatcher when it gets started. It is this registered IP address that should be specified as the “Coordinator IP address” for a part if that part is emulated by this emulation machine.

The “Emulation Machine IP address” field specifies the IP address of the above emulation machine. If the emulation machine has multiple IP addresses (i.e., it has multiple network interfaces), the address chosen must be the address of the network interface that connects to this real router. If an emulation machine has only one network interface, the “Coordinator IP address” and the “Emulation Machine IP address” for this machine will be the same. Finally, the “IP address of the real router’s interface” field specifies the IP address of the interface of the real router that corresponds to this port. These settings are for generating filtering and address translation rules for the emulation module in the kernel. The meanings of these fields will become clearer in Section V where several usage examples are provided.

If the virtual router is used to represent a crossover cable (or a layer-2 switch), one should tick the “Let multiple emulation machines communicate with each other directly” option to enable the bottom panel. On the this panel, each entry in the table is composed of four fields: Port ID, Assigned IP address, Coordinator IP address, and Emulation machine IP address. Their meanings are the same as their counterparts on the top panel. However, the “IP address of the real router’s interface” field, which is used in the top panel, is not used in the bottom panel as such information is not applicable in this configuration.
These two options can be supported by the same NCTUns emulation module. For both options, the processing in the module is like that for supporting a single-machine external router emulation. For the first option, when a packet is sent from part 1 and passed to the virtual router, both of its source and destination IP addresses will have been changed by the emulation module to the 200.Z.X.Y format (which has been presented in III-C). Like in the external router emulation design, the packet will be routed to the machine that emulates part 2 (i.e., the destination machine). For the second option, the performed address translations are exactly the same as above. That is, the addresses of the packet are also changed by the emulation module to the 200.Z.X.Y format before the packet leaves the machine that emulates part 1 (the source machine). The only difference is that in the second option the packet is passed directly to the destination machine rather than a real router.

Each of the two options has its own advantages. If a virtual router represents a real router, then packets generated by realworld applications can pass through them. Like the advantage provided by external router emulations, the functions of the real router can be tested by these real-world packets. On the other hand, if a virtual router represents a crossover cable, the cost of performing a distributed emulation is very low. Because each option has its own merits, NCTUns supports both options.

After the topology is divided into several parts, the GUI generates a suite of configuration files for each part and then sends each suite to an emulation machine that emulates the corresponding part. For an emulation machine, the coordinator program running on it will receive a suite and then fork a NCTUns simulation engine process to emulate the assigned part. In a distributed emulation, all emulation machines should emulate their assigned parts in real time and start their emulations at the same time. To achieve these goals, the NTP (Network Time Protocol) is used to synchronize the system clocks of all emulation machines and each NCTUns simulation engine process synchronizes its virtual clock with the system clock of its emulation machine every 1 ms. Because the transfer time of a suite to different emulation machines may take different amounts of time due to different network conditions, the GUI coordinates the start of these emulations by sending a “StartEmulation” command to each emulation machine with its starting time set to 5 seconds into the future. Later on, all emulation machines will start their emulations at the same time when this starting time is reached.

5. Usage Examples

5.1. Example 1

As shown in Fig. 8, a network is composed of two hosts and a virtual router. The virtual router divides the network topology into two parts. The host on the left part of the network transmits TCP packets to the host on the right part of the network. In the emulated network, the IP addresses assigned to these two hosts are 1.0.1.1 and 1.0.2.1, respectively, and the IP addresses assigned to the two ports of the virtual router are 1.0.1.2 and 1.0.2.2, respectively. One can see that the subnet on the left is allocated the 1.0.1.X addresses while the subnet on the right is allocated the 1.0.2.X addresses.
As explained previously, the virtual router can represent (1) a real router, or (2) a layer-2 switch or a crossover Ethernet cable. In the following, we first show the detailed steps for configuring a distributed emulation where the virtual router represents a real router.

A. Use a real router to connect emulation machines

As shown in Fig. 9, the physical setup contains two emulation machines and one real router. The IP addresses of these two emulation machines are 192.168.1.1 and 192.168.2.1, respectively, and these two emulation machines connect to the real router at the two interfaces with 192.168.1.254 and 192.168.2.254, respectively. The central controller (the GUI) and dispatcher are run on the emulation machine assigned the IP address 192.168.1.1 to save the number of machines that need to be used.

On the real router, we execute the following route commands:

```
route add -net 200.2.1.0/24 gw 192.168.1.1
route add -net 200.1.2.0/24 gw 192.168.2.1
```

The rationale for executing these commands on the real router is explained here. Remember that a packet with a destination IP address of 200.Z.X.Y indicates that before it leaves the emulated network to reach a real router, it is on the 1.0.Z.0/24 subnet and is destined to the 1.0.X.Y subnet. Therefore, the first route command will make the real router forward packets that leave the 1.0.2.0/24 subnet and destined to the 1.0.1.0/24 subnet to the emulation machine with the IP address of 192.168.1.1, which is responsible for emulating the 1.0.1.0/24 subnet. The second route command provides similar functions.
To set up the TCP sender and receiver applications that should be run up during emulation on the left and right hosts, in the GUI one can specify that the “stcp 1.0.2.1” command should be run on the left host and the “rtcp” command should be run on the right host, where “stcp” and “rtcp” are the names of real-world TCP sender and receiver applications freely available on the Internet. With these settings, these programs will be run up on the left and right emulation machines respectively to establish a real-world greedy TCP connection from the left host to the right host during emulation. To provide necessary information for the GUI to generate rules for the emulation kernel module, in the dialog box of the virtual router one needs to choose the first option “Use a real router to connect multiple emulation machines” and provide necessary information. The detailed settings for this example are shown in Fig.10.

![Virtual Router](image)

Figure 10. The settings for example 1 when the “Use a real router to connect multiple emulation machines” option is chosen.

**B. Use a direct link to connect two emulation machines**

If the virtual router does not represent a real router, it can represent a crossover link or a layer-2 switch. As shown in Fig.11, the two emulation machines are directly connected via a crossover Ethernet cable. Using this physical setup, the virtual router is not mapped to any device in the real world.

In this example case, the dispatcher and the GUI are run on the emulation machine with the IP address 192.168.1.1.

The “Let multiple emulation machines communicate with each other directly” option should be chosen for this physical setup. The detailed settings for this option are shown in Fig.12.
5.2. Example 2

In example 2, we use one virtual router to divide the network topology shown in Fig. 13 into three parts. Each part corresponds to a subnet in the emulated network. The part for subnet 1 (1.0.1.X) is on the top; the part for subnet 2 (1.0.2.X) is on the left; the part for subnet 3 (1.0.3.X) is on the right. The host with "1.0.1.1" IP address will establish a greedy TCP connection to the host with "1.0.2.1" IP address and the host with "1.0.2.2" IP address will establish a greedy TCP connection to the host with "1.0.3.1" IP address.

Similar to example 1, this example case also has two possible physical network configurations. One is using a real router to connect the three emulation machines and the other is using a layer-2 switch (which functions like a crossover cable) to connect the three emulation machines. The configurations and settings for these two cases are presented below.
A. Use a real router to connect emulation machines

As shown in Fig. 14, the physical setup is composed of three emulation machines and one real router. The IP addresses of these three emulation machines are 192.168.1.1, 192.168.2.1, and 192.168.3.1, respectively. These emulation machines connect to the real router at its three interfaces. The IP addresses of these interfaces are 192.168.1.254, 192.168.2.254, and 192.168.3.254, respectively. In this example, the dispatcher and the GUI are run on the emulation machine with the IP address 192.168.1.1.
route add -net 200.1.2.0/24 gw 192.168.2.1  
route add -net 200.1.3.0/24 gw 192.168.3.1  
route add -net 200.2.1.0/24 gw 192.168.1.1  
route add -net 200.2.3.0/24 gw 192.168.3.1  
route add -net 200.3.1.0/24 gw 192.168.1.1  
route add -net 200.3.2.0/24 gw 192.168.2.1

Because the rationale for executing these commands on the real router has been explained before, their meanings are not explained here again. Fig. 15 shows the detailed settings for example 2 when the "Use a real router to connect multiple emulation machines" option is chosen.

![Settings for example 2 when the "Use a real router to connect multiple emulation machines" option is chosen.](image)

**Figure 15.** The settings for example 2 when the "Use a real router to connect multiple emulation machines" option is chosen.

**B. Use a direct link to connect two emulation machines**

An alternative to connect the three emulation machines is to connect each pair of them by a direct link. However, connecting all of them to a layer-2 switch achieves the same goal and is more convenient. This physical setup is shown in Fig. 16.

In this example case, the dispatcher and the GUI are run on the emulation machine with the IP address 192.168.1.1. The "Let multiple emulation machines communicate with each other directly" option should be chosen for this physical setup. The detailed settings for this option are shown in Fig. 17.
5.3. Example 3

In example 3, we demonstrate a distributed emulation that uses two virtual routers to divide the network topology into three parts. Fig. 18 shows this example network. Each virtual router connects with two subnets and the 1.0.2.0/24 subnet on the middle is connected to both virtual routers. In this example case, the host 1.0.1.1 (in part 1 on the left) will establish a greedy TCP connection to the host 1.0.3.1 (in part 3 on the right) and the host 1.0.3.1 will establish a greedy TCP connection to the host 1.0.2.1 (in part 2 on the middle).
A. *Use a real router to connect emulation machines*

In this setup, each virtual router is mapped to a real router and therefore we use two real routers to connect three emulation machines together. As shown in Fig. 19, the IP address of the emulation machine for emulating part 1 is 192.168.1.1 and that for part 3 is 192.168.4.1. The emulation machine for emulating part 2 on the middle has two interfaces with IP addresses 192.168.2.1 and 192.168.3.1, respectively. Because its coordinator uses 192.168.2.1 to register with the dispatcher, later on when we configure the entries of the ports of the left and right virtual routers that connect to this part, we set all of their “Coordinator IP address” fields to 192.168.2.1.

As shown in Fig. 19, the IP addresses used on the real router that connects the emulation machines with 192.168.1.1 and 192.168.2.1 are 192.168.1.254 and 192.168.2.254, respectively; the IP addresses used on the real router that connects the emulation machines with 192.168.3.1 and 192.168.4.1 are 192.168.3.254 and 192.168.4.254, respectively. In this example, the dispatcher and the GUI are run on the emulation machine with the IP address 192.168.1.1.
Two types of route commands should be executed. The first type is for setting up the communication between the central controller (the GUI) and each coordinator in the real world. These route commands are shown below:

On the router with IP address 192.168.1.254 and 192.168.2.254:

route add -net 192.168.3.0/24 gw 192.168.2.1
route add -net 192.168.4.0/24 gw 192.168.2.1

On the router with IP address 192.168.3.254 and 192.168.4.254:

route add -net 192.168.1.0/24 gw 192.168.3.1
route add -net 192.168.2.0/24 gw 192.168.3.1

On the emulation machine with IP address 192.168.2.1 and 192.168.3.1

route add -net 192.168.1.0/24 gw 192.168.2.254
route add -net 192.168.4.0/24 gw 192.168.3.254

The second type is for setting up the communication between hosts in different parts of the emulated network. It is like what we did for example 1 and example 2. These route commands are shown below:

On the router with IP address 192.168.1.254 and 192.168.2.254:

route add -net 200.1.2.0/24 gw 192.168.2.1
route add -net 200.1.3.0/24 gw 192.168.2.1
route add -net 200.2.1.0/24 gw 192.168.1.1

On the router with IP address 192.168.3.254 and 192.168.4.254:

route add -net 200.2.3.0/24 gw 192.168.4.1
route add -net 200.3.1.0/24 gw 192.168.3.1
route add -net 200.3.2.0/24 gw 192.168.3.1

The route settings for this example are more complex than those used in the previous two examples. In this example case, the emulation machine on the middle needs to forward packets for other emulation machines. However, in the previous two example cases, this is unnecessary. For example, a packet from the host 1.0.1.1 to the host 1.0.3.1 needs to go through the subnet 1.0.2.0/24 to arrive at its destination node 1.0.3.1. This packet not only needs to be forwarded by the two real routers, but also needs to be forwarded by the emulation machine on the middle.

The detailed settings for the left and right virtual routers are shown in Fig. 20 and Fig. 21, respectively. Notice port 2 in the dialog box of the left virtual router and port 1 in the dialog box of the right virtual router. They both connect to the middle part. Because the coordinator running on the middle emulation machine uses 192.168.2.1 to register itself with the dispatcher, both of their “Coordinator IP address” should be set to 192.168.2.1. Regarding the
“EmulationMachine IP address” field, the settings for these two ports are derived based on the definition given in Section IV-B. For port 2 of the left virtual router, it is the IP address of the network interface on the middle emulation machine that connects to this left virtual (real) router — which is 192.168.2.1. For port 1 of the right virtual router, it is the IP address of the network interface on the middle emulation machine that connects to this right virtual (real) router — which is 192.168.3.1. These IP addresses are used as the gateway IP addresses in the generated route commands.

Figure 20. The settings for the left virtual router in example 3 when the “Use a real router to connect multiple emulation machines” option is chosen.

Figure 21. The settings for the right virtual router in example 3 when the “Use a real router to connect multiple emulation machines” option is chosen.
B. Use a direct link to connect two emulation machines

If two emulation machines are connected via a layer-2 switch (which can function like a crossover cable), they need to be on the same subnet. As shown in Fig. 22, the IP addresses of the emulation machines that emulate the left part and the right part of the network are 192.168.1.1 and 192.168.2.2, respectively. The emulation machine that emulates the middle part of the network has two interfaces with the IP addresses 192.168.1.2 and 192.168.2.1, respectively. In this physical setup, the left switch connects the two interfaces with 192.168.1.1 and 192.168.1.2 IP addresses and the right switch connects the two interfaces with 192.168.2.1 and 192.168.2.2 IP addresses.

![Diagram](image)

Figure 22. The physical setup when two layer-2 switches are used for example 3.

![Table](image)

Figure 23. The settings for the left virtual router in example 3 when the “Let multiple emulation machines communicate with each other directly” option is chosen.
Figure 24. The settings for the right virtual router in example 3 when the “Let multiple emulation machines communicate with each other directly” option is chosen.

In this example, the dispatcher and the GUI are run on the emulation machine with the IP address 192.168.1.1. The detailed settings for the left and right virtual routers are shown in Fig. 23 and 24, respectively. Notice port 2 in the dialog box of the left virtual router and port 1 in the dialog box of the right virtual router. Their “Coordinator IP address” fields should be set to 192.168.1.2. However, their “Emulation Machine IP address” fields should be set to 192.168.1.2 and 192.168.2.1, respectively.

6. Performance Evaluation

In this section, we conduct a complex distributed emulation case to show the performances and scalability of the NCTUns distributed emulation approach. As shown in Fig. 25, the topology is composed of 6 emulated routers, 6 virtual routers each is connected with 2 emulated routers. 12 emulated hosts as UDP senders and receivers, and 3 external hosts as UDP senders and receivers. This topology is divided into 6 parts. Among them, there are 3 parts each with one external host. These external hosts are real-world devices acting as UDP senders and receivers.

Each part is emulated by one emulation machine (EM) and each external host is a notebook computer. The following lists the specifications of these machines:

EM 1 - Intel Core 2 Duo E8400 (3.00 GHz), 2 GB RAM, with two 1 Gbps Ethernet NICs and one 100 Mbps NIC.
EM 2 - Intel Core 2 Quad Q9300 (2.50 GHz), 4 GB RAM, with two 1 Gbps Ethernet NICs.
EM 3 - Intel Core 2 Duo E8400 (3.00 GHz), 2 GB RAM, with two 1 Gbps Ethernet NICs and one 100 Mbps NIC.
EM 4 - Intel Core 2 Quad Q9300 (2.50 GHz), 4 GB RAM, with two 1 Gbps Ethernet NICs.
EM 5 - Intel Core 2 Duo E8400 (3.00 GHz), 2 GB RAM, with two 1 Gbps Ethernet NICs and one 100 Mbps NIC.
EM 6 - Intel Core 2 Duo E8400 (3.00 GHz), 2 GB RAM, with two 1 Gbps Ethernet NICs and one 100 Mbps NIC.

Each EM is a desktop PC and is connected with its two neighboring EMs on the hexagon using its two 1 Gbps Ethernet network interfaces. The 3 notebook computers use their 100 Mbps network interfaces to connect to the hexagon at EM 1, 3, and 5, respectively. The way the topology is divided is depicted in Fig. 26.

On the emulated network topology, the bandwidth of the links on the backbone hexagon is set to 1 Gbps while the bandwidth of all edge links (that connects an emulated or external host to the backbone) is set to 100 Mbps. The signal propagation delay of all links on the emulated network topology is set to 1 microsecond. We set up several UDP constant-bit-rate (CBR) traffic flows and they are categorized into two types: the traffic generated from the real world (i.e., from an external host) and the traffic generated from the emulated network.

Figure 25. The topology used in performance evaluations.
(i.e., from an emulated host). For the first type of traffic, Fig. 27 shows that there are 3 traffic flows each starting from an external host in part (i), passing through part (i+1), and then ending at an external host in part ((i+2) mod 6), where i is 1, 3, and 5. The data rate of these UDP flows is fixed to 1 Mbps across all cases. For the second type, Fig. 28 shows that there are 6 traffic flows each starting from an emulated host in part (i) and ending at an emulated host in part ((i mod 6) + 1), where i = 1, 2, 3, 4, 5, and 6. To evaluate the performances and scalability of NCTUns distributed emulation approach, the data rate of these flows are set to the same and is varied to generate different levels of traffic load. In this evaluation, the data rate is set to 1, 2, 5, 10, 20, 50, 60, 70, 75, 80, 90, 100 Mbps, respectively.

We expect that as the traffic load increases, the emulation machines participating in the distributed emulation will become slower to emulate its part and process the packets that flow in its part. When an emulation machine’s CPU power is insufficient to process a certain level of traffic load in real time, it will start to cause unnecessary processing delays to packets. In this situation, the generated results and packet logs will start to become less accurate and the distributed emulation has reached its performance limit.
Figure 27. The traffic generated from the real world.

Figure 28. The traffic generated from the emulated network.
To see where the performance limit is on these used PCs and notebook computers, we use measured round-trip delays encountered by ping packets over the emulated network as an indicator. To get these delay results, we let the sending nodes in Fig. 28 also send ICMP ping packets to their receiving nodes every 1 second. In our settings, because the bandwidth of the links on the hexagon is set to 1 Gbps and the bandwidth of the links that connect emulated and external hosts to the hexagon is set to only 100 Mbps, there is no congestion on the hexagon. Therefore, a ping packet should never experience any queuing delay on the hexagon. This means that if the CPU power of these emulation machines is sufficient, the measured round-trip delays of these ping packets should only contain the round-trip delay caused by packet transmission time and signal propagation delay on the links between their sending and receiving nodes. Given that the signal propagation delay of all links in this topology is set to only 1 microsecond and the transmission time of a 48-byte ping packet on these 1 Gbps or 100 Mbps links are less than 4 microseconds, the round-trip delay of these ping packets should be less than 1 millisecond if the participating emulation machines can still handle the traffic load. For this reason, we use the average round-trip delay of all ping packets as the performance indicator. When it starts to grow quickly, we know that the performance limit has been reached.

Fig. 29 shows the average round-trip delay of ping packets under different traffic loads. When the traffic load is less than 80 Mbps, we see that the average delay is less than 1 ms, which means that the participating emulation machines can still perform the distributed emulation very accurately. When the load goes up to 90 Mbps, we see that the offered load has exceeded the capacity of emulation machines and the average delay grows very quickly to a very large value. This shows the performance limit of the NCTUns distributed emulation approach on these tested machines.

![Figure 29. Average round-trip delay of ping packets between emulated hosts.](image)

Another useful performance indicator is the packet loss rate of all traffic flows during the distributed emulation. As explained before, due to the link bandwidth settings for both the physical links and emulated links, there is no congestion on either the emulated network or the physical network. As a result, no packet should be dropped due to congestion when they
traverse over the emulated network. However, when the offered traffic load goes beyond what the CPUs of these emulation machines can handle, the CPU of an emulation machine will become too busy to receive and process packets sent from other emulation machines in real time. When its input buffer is full and no longer can accommodate more incoming packets, packets will start to be dropped when they are transmitted between emulation machines, and this will cause packet losses to all traffic flows in this case. For this reason, we also use the average packet loss rate of all traffic flows as a performance indicator. When it starts to grow quickly, we know that the performance limit has been reached.

Figure 30. Average packet loss rate of all traffic flows

Fig. 30 shows the average packet loss rate of all traffic flows under different traffic loads. We see that the packet loss rate is 0% when the traffic load is less than or equal to 80 Mbps. When the load goes up to 90 Mbps, the packet loss rate starts to grow very quickly, which
means that 90 Mbps is the maximum traffic load that the used emulation machines can handle for this distributed emulation case. This observation is consistent with the average round-trip delay results presented above.

Fig. 31 shows the average CPU usage of emulation machine 1 under different traffic loads. As expected, the trend is that the CPU usage increases as the load increases. We see that when the load is less than or equal to 80 Mbps, the CPU usage is less than 100%, which means that the CPU can still handle these loads in real time. However, when the load goes up to 90 Mbps, the CPU usage starts to stay at 100%, which means that it has used up all of its CPU cycles to handle the load. This result is consistent with the packet delay and packet loss results shown above as these performance indicators start to worsen at 90 Mbps.

![Figure 32: Using only one emulation machine to emulate the whole network.](image)

As presented in the Introduction Section, one advantage of distributed emulation is that, under the real-time emulation criteria, it can emulate a larger network than a single machine can do. That is, performance speedup is one of its advantages. To measure the performance speedup, we make a single-machine emulation case in which only one emulation machine is used to emulate the whole network and connect with three external hosts. Compared with the distributed emulation case, the six virtual routers now are replaced with six emulated routers. However, the settings for link bandwidth, link delays, traffic loads, etc. are all the same as those used in the distributed emulation case. Fig. 32 shows this configuration.

For this single-machine emulation, we measure how many seconds in the real world are needed to advance a second in the emulation under different traffic loads. That is, the performance speedup of distributed emulation over singlemachine emulation. Fig. 33 shows
that using the distributed emulation approach, only one second is required to advance one second in the emulation under 80 Mbps traffic load (i.e., the real-time emulation criteria is still met). However, it shows that as the traffic load increases, it takes longer and longer to advance a second in the emulation if the traditional single-machine emulation approach is used. If we look at the load of 80 Mbps, where the distributed emulation approach is still useful, we see that a performance speedup of 3.9 is achieved when 6 emulation machines work together to perform this distributed emulation. This demonstrates the performance speedup advantage of the NCTUns distributed emulation approach.

Figure 3. The time used to advance a second in emulation time (i.e., performance speedup).

7. Conclusion

In this paper, we design a distributed emulation approach and implement it on the NCTUns network simulator and emulator. We present its internal design and implementation and illustrate how the source and destination IP addresses of a packet are translated in a distributed emulation. A distributed emulation is carried out by multiple emulation machines working together at the same time. It provides several important advantages such as the ability to emulate a very large network in real time and overcomes the hardware limitations when only a single machine is used to carry out an emulation.

In addition, we present various usages of this approach, showing its flexibility and capability in emulating different network topologies. To show its performances and scalability, we conduct a complex large network emulation using this approach. The measured packet delay and loss results show that on the tested PCs and notebook computers, a performance speedup of 3.9 can be achieved when 6 emulation machines work together to carry out the distributed emulation.
References


