Integrated Network Experimentation using Simulation and Emulation

Shashi Guruprasad Robert Ricci Jay Lepreau

School of Computing, University of Utah
{*shash,ricci,lepreau*}*@cs.utah.edu*

Abstract

Discrete-event packet-level network simulation is well-known and widely used. Network emulation is a hybrid approach that combines real elements of a deployed networked application—such as end hosts and protocol implementations—with synthetic, simulated, or abstracted elements—such as the network links, intermediate nodes and background traffic. A key difference between the two approaches is that in the former, the notion of time is virtual and is independent of real time, whereas the latter must execute in real time. Emulation gains realism while naturally foregoing complete repeatability; historically, emulation was also tedious to control and manage.

We define integrated network experimentation as spatially combining real elements with simulated elements in the same experimental run, each modeling different portions of a network topology. Integrated experiments enable new validation techniques and larger experiments than obtainable using real elements alone. This paper highlights the key issues in integrated network experimentation, and presents some of the design techniques we use in designing, building, and putting into public production use such an integrated environment, running on a space-shared cluster.

1. Introduction

There are three experimental techniques used in the design and validation of new and existing networking ideas: simulation, emulation and live network testing. All three techniques have unique benefits and drawbacks. However, they need not be viewed as competing techniques—using more than one technique can help validate ideas better than using any one technique alone.

Network simulation provides a repeatable and controlled environment for network experimentation. It is easy to configure and allows a protocol to be constructed at some level of abstraction, making simulation a rapid prototype-and-evaluate environment. Ease of use also allows for exploration of large parameter spaces.

Network emulation [18, 25, 22, 8] is a hybrid approach that combines real elements of a deployed networked application—such as end hosts and protocol implementations—with synthetic, simulated, or abstracted elements—such as the network links, intermediate nodes and background traffic. Which elements are real and which are partially or fully simulated will often differ, depending on the experimenter's needs and the available resources. A fundamental difference between simulation and emulation is that while the former runs in virtual simulated time, the latter must run in real time. Another important difference is that it is impossible to have an absolutely repeatable order of events in an emulation. That is due to its realtime nature and, often, a physically-distributed computation infrastructure.

We define integrated network experimentation as *spatially* combining real elements with simulated elements, each modeling different portions of a network topology in the same experimental run. Integrated experimentation leverages the advantages of using real and simulated elements, providing valuable capabilities. It enables i) validation of experimental simulation models against real traffic loads; ii) validation of real applications against repeatable, congestion-reactive cross traffic derived from a rich variety of existing, validated simulation models; iii) by multiplexing simulated elements on physical resources, scaling to larger topologies than would be possible with real elements only; and iv) direct interaction of real applications with arbitrary simulated networks.

A related form of network experimentation is to integrate experimental techniques *temporally*, as researchers experiment iteratively on the same input, providing comparison and validation. While the advantages of the latter are discussed elsewhere [25], this paper is confined to discussing issues in spatial integration.

In this paper we outline the key issues in integrated network experimentation and present some of our design techniques. The latter include automatic partitioning of the network topology specification, scalable resource-conserving assignment of physical resources to the topology, and

feedback-directed auto-adaptation of the resource assignment, to ensure that simulators run in realtime. We demonstrate integrated network experiments by seamlessly integrating simulated resources via *nse* [8] in the Emulab testbed [25]. Our work in automatically partitioning topologies across physical PCs and switches also benefits "pure" distributed network simulation, although we have not yet implemented that.

The rest of the paper is organized as follows. In Section 2, we briefly describe Emulab and *nse*. Section 3 describes how we automate integrated experiments. In Section 4 we go into the details of scalable resource assignment. In Section 5, we explain our algorithm for performing auto-adaptation, including two heuristics. Section 6 outlines a few results. We discuss related work in Section 7, how our work is applicable to "pure" distributed simulation in Section 8, and we then conclude.

2. Testbed Context

The Emulab software is the management system for a network-rich PC cluster that provides a space- and time-shared public facility for studying networked and distributed systems.

An "experiment" is Emulab's central operational entity. An experimenter first submits a network topology specified in an extended *ns* syntax. This virtual topology can include links and LANs, with associated characteristics such as bandwidth, latency, and packet loss. Limiting and shaping the traffic on a link, if requested, is done by interposing "delay nodes" between the endpoints of the link, or by performing traffic shaping on the nodes themselves. Specifications for hardware and software resources can also be included for nodes in the virtual topology.

Once the testbed software parses the specification and stores it in the database, it starts the process of "swapin" to physical resources. Resource allocation is the first step, in which Emulab attempts to map the virtual topology onto the PCs and switches with the three-way goal of meeting all resource requirements, minimizing use of physical resources, and running quickly. In our case the physical resources have a complex physical topology: multiple types of PCs, with each PC connected via four 100 Mbps Ethernet interfaces to switches that are themselves connected with multi-gigabit links. The testbed software then instantiates the experiment on the selected machines and switches. This can mean configuring nodes and their operating systems, setting up VLANs to emulate links, and creating virtual resources on top of physical ones. Emulab includes a synchronization service as well as a distributed event system through which both the testbed software and users can control and monitor experiments.

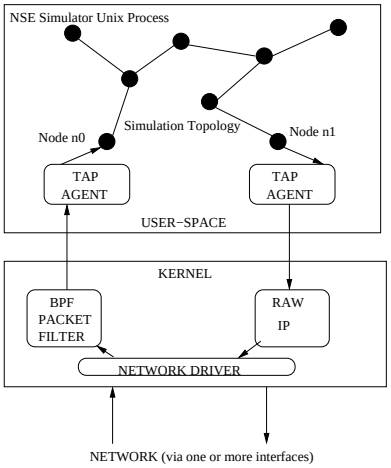


Figure 1. The basic operation of the *ns* emulation facility (*nse*)

The simulation back-end in Emulab that is described in the rest of the paper takes advantage of the *ns* emulation facility (called *nse*) [8] that permits simulated packets to leave the simulator for the "real" network, and vice versa.

We define here some of the terms we use in the rest of the paper. A *pnode* is a physical PC node in Emulab. A virtual topology is one that an experimenter specifies and is independent of its physical realization in the experimental environment. A *vnode* is a node in the virtual topology which could be of different types, such a PC vnode, simulated vnode or "virtual machine" vnode. Similarly, *vlinks* and *plinks* are virtual links in the virtual topology and physical links respectively.

3. Automation

Automation of the complete experiment life-cycle can be a big factor in the overall experimentation time. For example, comparisons between manual experimental setup against an automated one of a 6-node "dumbbell" topology in Emulab show an improvement of a factor of 70 in the automated case [25]. The user of Emulab specifies a integrated experiment in *ns*-like OTcl code that is agnostic to how the experiment is physically realized. A user encloses *ns* code in a `make-simulated` block to indicate which portions of the specification are to be simulated. There can be several such blocks in a single specification which are eventually concatenated. One of the aspects of automating the mapping of the simulated portion of an integrated experiment is that the specification of the simulated topology must be used to generate multiple specifications for each sub-portion of the topology that gets mapped to different physical nodes. The structure of the specification language and the relationship

between virtualized resources can have an impact on the ease of doing this. The use of OTcl, a general purpose programming language with loops and conditionals, for specification in *ns* and Emulab makes the above task difficult compared to a domain specific language that enforces relationships between resources at the syntactic level. For example, the domain specific language (DML) used by the scalable simulation framework (SSF [6]) simulator has a hierarchical attribute tree notation [5] that would make it easier to generate specifications of sub-portions of the full topology. DML is a static, tree-structured notation similar to XML. Because of the tree-structured nature of DML, a node and its attributes as well as any resources belonging to the node are syntactically nested. Thus, a simple parser can partition such a node and everything associated with it easily. On the other hand, *ns* OTcl can have simulated resources with logically nested relationships scattered anywhere in the code without explicit syntactic hints, making such partitioning more complex.

Emulab uses an OTcl interpreter to parse a user's OTcl specification into an intermediate representation and stores this in a database [25]. The parser statically evaluates the OTcl script and therefore takes loops and conditionals into account. Using a similar approach, we have developed a custom parser to parse the simulated portion of the integrated experiment specification and generate new OTcl specifications for each sub-portion of the topology mapped to a physical node [1]. We have extended the Emulab parser to store the simulated part of the experiment specification (enclosed in one or more `make-simulated` blocks) into the database "as is" for further parsing later. This is necessary since OTcl sub-specifications can be generated only after the mapping phase. We will call this second parse as *nse parse*. Our approach for the *nse* parse is similar to Emulab's initial parsing. The output of the *nse* parse is a set of OTcl sub-specifications that are targeted to different simulator instances. Once generated, these sub-specifications are stored in Emulab's database to be used during the experimental run. We describe our approach for this parse below.

Written in OTcl, the parser operates by overriding and interposing on standard *ns* procedures. Some key *ns* methods are overloaded. These methods use the mapping information from Emulab's database to partition the user-specified OTcl code into OTcl sub-specifications for each simulator instance. Due to the structure of classes in *ns*, we are able to support a large portion of *ns* syntax in the `make-simulated` block. *ns* classes that are important for this parsing phase are `Simulator`, `Node`, `Agent` and `Application`. Links in *ns* are typically instantiated using the `duplex-link` method of the `Simulator` class. Traffic in *ns* has a simple two layer model: transport and application. Subclasses of the `Agent` class normally implement the transport layer functionality of any protocol. These agents are all attached to a `Node` object. Similarly, subclasses of the `Application` class implement the application layer functionality of a protocol. The application objects are all attached to some agent. `Agent` and `Application` are thus directly or indirectly associated with a `Node` object, allowing the OTcl code for them all to be generated for a particular simulator instance. All simulation objects support the specification of per-object attributes via instance variables.

Classes in *ns* have structured names with the hierarchy delineated by the slash (`/`) character. For example, all subclasses of the `Agent` class have a `Agent/` prefix. Tcl/OTcl also supports `info` procedures that can help extract the state of the OTcl interpreter Similarly, an `unknown` method permits us to capture arbitrary method calls without any knowledge about them. Using the above features, we are able to reconstruct the OTcl code to be given to different instances of *nse*. Note that most of the constructs are re-generated as they were specified by the experimenter while others such as links are transformed into `rlinks`, a type of link we have created, if the incident nodes are mapped to different simulator instances. These `rlinks` perform encapsulate simulator packets in IP packets and inject them into the network. At the other end, we perform packet capture, de-encapsulate the IP packet into a simulator packet and introduce it into the simulation. IP addresses are used for the global routing of packets. Emulab automatically assigns IP addresses and generates global shortest path routes which we then use during the simulation. For user-specified events specified with the `at` method of the `Simulator` class, our overridden `at` method makes an attempt to map the event to a simulation object which is further mapped to the simulator instance to which it needs to be delivered. The events thus mapped are thus stored in the database. OTcl code is not generated for user-specified events since they will be delivered via Emulab's central event scheduler as described in the technical report [10]. The following points outline a list of steps performed by the *nse* parser:

1. Concatenate all `make-simulated` blocks and store it in the database during first parse along with topology info.
2. Perform mapping using topology information from the database.
3. Initialize mapping info from the database in the *nse* parser (OTcl based).
4. Source the code in `make-simulated` block into the parser creating objects based on overridden classes such as Simulator, Node, Agent etc. that we have defined. The base class object is created for any class

1 The implementation described in this section can be re-targeted to generating OTcl sub-specifi cations to map pure distributed simulation using *pdns*, albeit with modest changes to our implementation. We hope to complete that separate project in the medium future.

name with a prefix of one of the above special classes. The actual name of the subclass is stored and will be used later to re-generate Tcl code. Objects of unrecognized classes are ignored.

5. Unrecognized global Tcl procedures are ignored. Note that unrecognized methods for the special classes mentioned above are all captured using the `unknown` method for the respective classes.

6. The last step is to generate OTcl code in this order of simulation objects: `Simulator`, `Node`, `duplex-link`, `rlink`, `Agent`, `Application`. The code generated will have method invocations as well as initialization of instance variables.

While our approach works well within the bounds of careful specification, numerous counter-examples of experiment specification can be constructed where our approach for parsing when using either Emulab frontend parser or *nse* parser will fail or is not adequate. For example, if specified code had dependencies on variable values of a running simulator, our approach fails. Another example is if an experimenter specified an OTcl loop to create a large number of simulation objects, our code generation will unroll all the loops, potentially causing code bloat that may be beyond the limits of our system. Some of these limitations can be overcome with more work while others are difficult to do so without writing a complete interpreter that understands all of *ns*. Despite these limitations, we have run automated integrated experiments with a simulated topology of 400+ nodes and 400 simulated FTP/TCP traffic flows.

4. Scalable Resource Assignment and Mapping

Network experimentation on real hardware requires a mapping from the virtual resources an experimenter requests to available physical resources. This problem arises in a wide range of experimental environments, from network emulation to distributed simulation. This mapping, however, is difficult, as it must take a number of varying virtual resource constraints into account to "fit" into physical resources that have bandwidth bottlenecks and finite physical node memory and compute power. Poor mapping can reduce efficiency and worse, introduce inaccuracies—such as when simulation events cannot keep up with real-time—into an experiment. We call this problem the "network testbed mapping problem" [16]. In general graph theory terms, this is equivalent to the graph embedding or mapping problem with additional constraints specific to this domain. This problem is NP-hard [16].

The mapping could be many-to-one, such as multiple vnodes and vlinks on a physical node, one-to-one, such as a router node on a physical PC, or one-to-many, such as a vlink of 400Mbps that uses four physical 100Mbps links [2]

When simulated traffic interacts with real traffic, it must keep up with real time. For large simulations, this makes it necessary to distribute the simulation across many nodes. In order to do this effectively, the mapping must avoid "overloading" any pnode in the system, and must minimize the links in the simulated topology that cross real plinks. By "overload," we mean that there are more simulated resources mapped to a pnode than the instance of a simulator can simulate in real-time.

"Pure" distributed simulation also requires similar mapping. In this case, rather than keeping up with real time, the primary goal is to speed up long-running simulations by distributing the computation across multiple machines [4]. However, communication between the machines can become a bottleneck, so a "good" mapping of simulated nodes onto pnodes is important to overall performance. While this is achieved by minimizing the number of vlinks that cross pnodes, another factor that affects performance is the *lookahead* that can be achieved. Lookahead refers to the ability to determine the amount of simulated time that could be safely processed in one simulator process without causality errors due to events from a different simulation process. Lookahead is affected by the *distance* between simulation processes [9]. Distance provides a lower bound in the amount of simulated time that must elapse for an unprocessed event on one process to propagate to (and possibly affect) another process. Therefore, it is not just important that a good mapping has fewer links crossing simulation processes, but also for them to be low bandwidth and/or high latency links because they increase distance and thus lookahead, leading to improvement of efficiency of a distributed simulation.

A good mapping has the following properties:

- **Sparse cuts**: The number of vlinks whose incident vnodes are mapped to different pnodes should be minimized. At the same time, the number of vnodes and vlinks mapped to the same pnode should not exceed its emulation capacity.
- **Low congestion**: The number of vlinks that share plinks should be minimized without over-subscribing the plinks. While some plinks such as node-to-switch plinks are dedicated to an experiment, others such as inter-switch plinks are shared between experiments. By minimizing vlinks mapped to shared plinks, space-sharing efficiency is increased.
- **Low dilation**: The physical length (i.e., hops) that correspond to mapped vlinks, also known as dilation, should be kept to a minimum. For example, a vlink

2 Emulab currently does not support bonding physical links to form vlinks of higher capacity than that of a single plink.

that is mapped to a plink that traverses multiple cascaded switches, is less desirable than one that traverses only one switch.

- **Efficient use of resources across experiments**: The unused capacity of physical resources that are not shared across experiments must be kept to a minimum. In other words, minimize usage of shared resources such as inter-switch links and maximize usage of experiment private resources such as pnodes and switch links from/to these nodes.
- **Fast Runtimes:** A sub-optimal solution arrived at quickly is much more valuable than a near optimal solution that has very long runtimes (e.g., minutes vs. hours). This aspect becomes important when we map iteratively using runtime information to perform auto-adaptation of simulated resources. Due to the NP-hard nature of the problem, the runtimes are easily exacerbated by larger topologies made possible by "soft" resources such as simulated or "virtual machine" resources.

`assign` [16] is the name of the resource assignment program we have developed in Emulab. It supports a node type system. Each node in the virtual topology is given a type by the experimenter, and each node in the physical topology has a set of types that it is able to satisfy. Each type on a pnode is associated with a "packing factor" (also known as "co-locate factor"), indicating how many nodes of that type it can accommodate. This enables multiple vnodes to share a pnode, as required by integrated experimentation as well as "pure" distributed simulation. For example, if `sim` is the type associated with simulated nodes, a pnode will support a co-locate factor for nodes of type `sim`. However, if all virtual or simulated nodes are considered to be equal, this can lead to sub-optimal mapping since typically the pnode resources consumed by vnodes are all different. To achieve better mapping, arbitrary resource descriptions for vnodes and pnodes need to be supported. However, this adds a bin-packing problem to an already complicated solution space. In order to flexibly support soft resources such as simulated or "virtual machine" resources, several new features were added recently to `assign` [11]. We describe these features below[3].

Limited Intra-node bandwidth: When multiple vnodes are mapped to a pnode, vlinks are also mapped to the same pnode. Originally, there was no cost for such vlinks which makes it possible to potentially map an arbitrarily large number of vlinks. In reality however, there is a limit on the number and total capacity of vlinks that can be supported. An idle vlink has no cost other than memory used up in the simulator. However, there is a computational cost of processing packets when traffic passes through vlinks. `assign` now supports an upper limit on the intra-node bandwidth and uses it when mapping vlinks whose capacities are allowed to add up to the bandwidth. When mapping simulated resources, we set this capacity to 100Mbps on Emulab hardware, based on measurements reported elsewhere [10].

Resource Descriptions: Pnodes support arbitrary resource capacity descriptions such as CPU speed, memory, measured network emulation capacity, and real-time simulation event rate. Essentially this could be any value that represents an arbitrary resource capacity. Capacities for multiple resources are possible per pnode. Thus, vnodes with resource usage values for multiple resources are counted against the above capacities. It is possible to use resource descriptions even if only relative resource usage between vnodes is known. For example, if vnode A consumes thrice as many resources as vnode B, vnode A when mapped to an empty pnode would become 75% full.

Dynamic Physical Equivalence Classes: `assign` reduces its search space by finding groups of homogeneous pnodes and combining them into physical equivalence classes. When multiplexing vnodes on pnodes, a pnode that is partially filled is not equal to an empty node. This is not just in pnode capacity but also in its physical connectivity to other pnodes since the plinks between them are also partially filled. `assign` now computes the physical equivalence classes dynamically while mapping. While this helps a little, this feature is close to not having physical equivalence classes at all. This factor is the dominant contributor to longer runtimes when mapping multiple vnodes on pnodes, compared to an equal-sized topology with one-to-one mapping. For large topologies, the runtimes can be very long, into the tens of minutes and even hours.

Choice of pnode while mapping: As we noted before, a good mapping is likely to map two vnodes that are adjacent in the virtual topology, to the same pnode. Instead of selecting a random pnode to map a vnode, `assign`, now, with some probability, selects a pnode to which one of the vnode's neighbors has already been assigned. This dramatically improves the quality of solutions, although not the runtimes on large topologies.

Coarsening the virtual graph: Using a multi-level graph partitioner, METIS [12], which runs much faster than `assign` primarily because it has no knowledge of the intricacies of the problem, the virtual topology is "coarsened". By "coarsening," we mean that sets of vnodes are combined to form a "conglomerate" to form a new virtual graph which is then fed to `assign`. This feature dramatically improves runtimes, again due to the reduction in search space, making it practical to perform auto-adaptation.

3 We discuss how they are used to iteratively map simulated resources in section 5.

5. Feedback-directed Auto-adaptation of Simulated Resources

A mapping of simulated resources to physical resources should avoid "overloading" any pnode in the system, which was discussed in detail in section 4. The workload to which an instance of *nse* is subjected is not easily determined statically in an integrated experiment, partly because an experimenter can generate arbitrary traffic without specifying its nature apriori. An overloaded pnode will result in simulation inaccuracies. In the case of simulated resources, these inaccuracies occur because the simulator is not able to dispatch all events in real-time. A similar issue also arises when multiplexing "virtual machine" type vnodes on pnodes. In order to solve this issue, we perform auto-adaptation of simulated resources when overload is detected. Successive mappings use feedback data from running the experiment with prior mappings, until no overload is detected or we run out of physical resources. Such a solution for "virtual machine" type vnodes is discussed elsewhere [11]. In this section, we focus on performing auto-adaptation of simulated resources.

The factors that make it feasible for us to perform auto-adaptation are:

Fast mapping: This was discussed in section 4. A mapping that takes hours is clearly too slow. Iterative mapping reduces the search space by re-mapping only the portions of the topology that were mapped to pnodes reporting an overload.

Fast pnode reconfiguration: Iterative mapping is affected by the speed of *reconfiguring* pnodes for the new mapping, both pnodes currently reserved to the experiment and new ones that may be allocated as more resources are needed. Current PC hardware can take long enough to boot that this starts to affect re-mapping time. Emulab in a recent optimization, now avoids doing full reboots by having unused pnodes wait in a "warm" state in the boot loader. This boot loader has the ability to boot into different disk partitions, and to boot different kernels within those partitions. Pnodes that were already part of the experiment are reconfigured without rebooting. This involves pushing all the Emulab client configuration data to the pnode, reconfiguring interfaces, routing tables, and a new simulation.

Initial mapping is guided by optimistic vnode co-locate factors per pnode type in Emulab. A more powerful PC supports a higher co-locate factor than a less powerful one. The co-locate factor is intended as a coarse grained metric for CPU and memory load on a pnode. In simulations with lots of traffic, the CPU bottleneck is typically reached much earlier than memory limits are reached. Also, if different amounts of traffic are passing through different vnodes, their resource consumptions will be different. Considering these problems, the co-locate factor we choose is only based on a pnode's physical memory. Based on feedback data obtained from running the simulations, we hope to quickly converge to a successful experiment if the initial mapping is too optimistic. A simulated vnode in *nse* consumes only moderate amounts of memory, allowing us to support a large co-locate factor. According to a study that compared several network simulators [7], *ns* allocated roughly 100KB per connection, where each connection consists of two nodes with two duplex-links that each add new branches to a "dumbbell" topology. Each connection consisted of a TCP source and sink on the leaves of the dumbbell. On an Emulab PC with 256–512MB of memory, a fairly large co-locate factor can be supported.

When an overload is detected by a simulator instance, it reports all necessary information to Emulab *masterhost* via the event system. On receiving the first such event, a program on the *masterhost* is run that waits for several seconds, giving sufficient time for other pnodes to report overload if present. This program stores the feedback data into the database and begins re-mapping the experiment.

We outline two heuristics that we separately experiment with to guide auto-adaptation:

Doubling vnode weights: A coarse heuristic that we use is to double the weight of all the simulated nodes hosted on the pnode that reported an "overload" and re-map the topology. These simulated nodes will then consume twice as many slots from the pnode co-locate factor as before. This process repeats until no overload is detected or a vnode is being mapped one-to-one to an overloaded pnode. If the overload is still present, it means that the experiment could not be mapped on Emulab hardware.

Vnode packet-rate: Simulation event-rate is directly proportional to the rate of packets that pass through a vnode or are generated by that vnode. This is because every packet typically causes roughly a constant number of events. For packet forwarding, even though events in *ns* occur in links, the cost of processing these events can be attributed to the vnode to which such links are connected. Because the Emulab mapper, `assign`, associates resource capacities with pnodes and resource use with vnodes, we use the rate of packets passing through a vnode as the cost. Based on packet-rate measurements for Emulab hardware that we have reported in a detailed technical report [10], we set the pnode packet-rate capacities. This is a fine-grained heuristic compared to the previous one. Starting from an optimistic mapping, we can easily identify the vnodes that are "heavyweight," allowing subsequent mappings to pack such vnodes less tightly.

6. Results

6.1. Validation of distributed *nse*

When simulated resources in an integrated experiment are mapped to multiple PCs, some of the flow endpoints also get mapped to different *nse* instances on different PCs. In order to determine how similar are packet flows inside a single instance of *nse* compared to the ones that cross physical (switched) links, we perform the following experiment:

The basic experimental setup consists of two simulated nodes connected by a T1-like duplex-link of 1.544Mbps bandwidth and 25ms latency. Traffic is generated using `Agent/TCP` which is an abstract implementation of the BSD Tahoe TCP protocol [2]. About 75MB of data are transferred over this connection in one direction. This gives us a trace of approximately $50,000$ data packets and approximately the same number of ACK packets in the reverse direction. The simple model described above is useful in establishing a lower bound on the difference between absolute repeatable simulations and emulations using distributed *nse*. (In the rest of this section, a "TCP-sink" is the endpoint which receives DATA packets, while a "TCP-source" refers to the endpoint that sends DATA packets and receives ACKs.)

The above setup is realized under the following scenarios:

1. Both simulated nodes are in one instance of *nse*, i.e., pure real-time simulation. Whenever we use *RTSIM* anywhere in this section, we mean this scenario. Unless *nse* falls behind real-time due to an intensive workload, these results are the same as that of pure simulation. We have verified that all RTSIM results reported in this section exactly match pure *ns* simulation (i.e., running in discrete virtual time) which runs faster than real-time for this workload.
2. Each simulated node is in a different instance of *nse* on two different PCs connected via a 100Mbps switched Ethernet link. Each instance of *nse* simulates one node and the outgoing link to the other node. The physical 100Mbps link is simply used as a transport for the encapsulated simulator packets[4]. We will refer to this scenario in this section by *DIST-RTSIM*.
3. The above scenario is replicated to have 60 simulated **T1** links mapped to the same 100Mbps switched Ethernet link. Each instance of *nse* is handling approximately 7646 packets per second which is within the stable capacity of *nse* on this hardware, as detailed in the technical report [10]. Note that these are encapsulated packets roughly about 100 bytes in size resulting in 6–7% utilization of a 100Mbps Ethernet link. The simulated nodes on each end for these links are mapped to two different *nse* instances running on two PCs. This setup is useful in identifying the effects of multiplexing packets from independent virtual links over the same physical link. We will refer to this scenario in this section as *DIST-RTSIM-60*.

Our platform for all tests is an 850Mhz Pentium-III PC, 512 MB DRAM, running FreeBSD 4.9.

We now present comparisons between RTSIM, DIST-RTSIM and DIST-RTSIM-60.

Aggregate throughput comparisons for all three cases above had very small percentage errors from the expected value ($< 0.02\%$). We present the comparison of frequency of packet inter-arrivals at the TCP-sink in figure 2. Several more data points for such comparisons including *time-variance* plots are available in the technical report [10].

7. Related Work

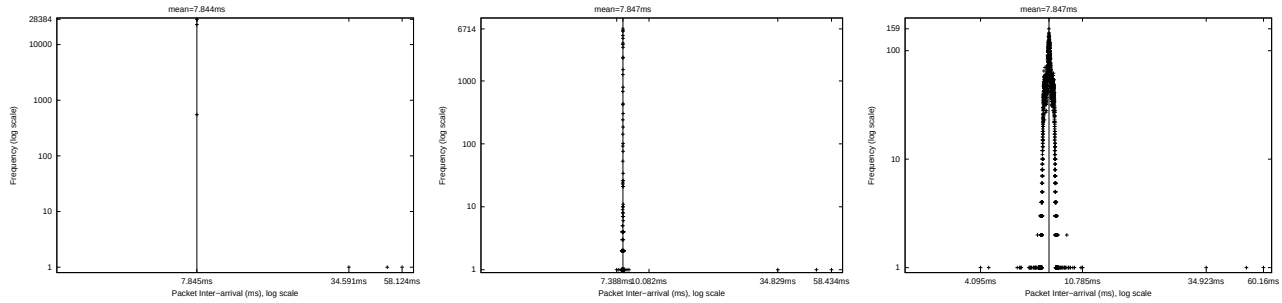
Modelnet [22] is an emulation system focused on scalability. It uses a small gigabit cluster, running a much extended and optimized version of Dummynet which is able to emulate an impressively large number of moderate speed links. It has the added capability of optionally distilling the topology to trade accuracy for scalability. It is complementary to our work. Our work leverages *ns*'s rich variety of models and protocols.

Dynamic Network Emulation Backplane [1] is an ongoing project that uses a dynamic library approach for capturing, synchronizing and re-routing network data from unmodified distributed applications over a simulated network. They also define an API for heterogeneous simulators to exchange messages, synchronize simulation time and keep pace with real time in order to simulate a larger network, thus leveraging the strengths of different network simulators. Time-synchronization in the distributed simulation case has a high overhead and it remains to be seen whether it can be performed in real-time. Data is captured from unmodified applications by intercepting system call functions using a dynamic-library pre-loading approach. This however, is platform dependent as well as error prone due to duplication of code and specialized implementation of several system calls.

nsclick [15] embeds the click modular router [14] in ns-2 which allows a single click based protocol implementation to run over a simulated wired or wireless network as well as on a real node on a real network.

NCTUns [24] is a TCP/IP simulator that has descended from the Harvard network simulator [23]. This simulator virtualizes the OS's notion of time to be the simulation

4 The size of the encapsulated packets does not always depend on the simulated packet size since packet data is not typically included. In the experiments performed here, the encapsulated packet size including the IP and Ethernet headers was about 100 bytes.



(a) RTSIM (b) DIST-RTSIM (c) DIST-RTSIM-60 (worst of 60 fbws)

time. Using a combination of link simulation and tunnel devices, a network topology is built on a simulation host machine. Applications and the protocol stack are unmodified and therefore more realistic than traditional simulators.

umlsim [3, 21] extends user-mode Linux (UML) with an event-driven simulation engine and other instrumentation needed for deterministically controlling the flow of time as seen by the UML kernel and applications running under it. The current level of support allows network experimentation with a single link. The degree of multiplexing of virtual nodes is limited due to the use of a complete kernel. The current system also performs poorly and has much scope for improvement.

The X-bone [19] is a software system that configures overlay networks. It is an implementation of the Virtual Internet Architecture [20] that defines "revisitation" allowing a single network component to emulate multiple virtual components, although in their context, a packet leaves a physical node before returning on a different virtual link. In our system, multiple routing tables as well as the context of a virtual link are needed even when all the nodes and links of a virtual topology are hosted on one physical host. In the general sense, however, the issues they identify have a broader scope than what they have restricted themselves to, in their paper. Thus, virtual internets can be formed not just in the wide-area but also using a cluster such as the one employed in Emulab. Integrated network experimentation spans all three experimental techniques and we therefore believe that it is the most comprehensive form of virtual internet that we know of.

8. Discussion

Users of a simulator such as pdns [17] have to manually partition a simulation model into its sub-models that run on different processors and are required to specify global addressing and routing to run the simulation. The DaSSFNet [6, 13] simulator partitions a simulation model using METIS [12]. However, the number of machines used for simulation has to be specified by the user and is used to determine the number of partitions. When performing only static partitioning, it is difficult to load-balance the simulation workload across all processors. A user typically has a fixed set of machines available for simulation that are not space-shared. Thus, there is no incentive to reduce the number of processors used even if the simulation can be performed nearly as efficiently on a smaller number of processors. In a space-shared environment, this becomes important.

The constraint for mapping a integrated experiment is that simulations must execute in realtime. In the case of "pure" distributed simulations, the goal is to reduce the runtime of long running simulations.

Acknowledgments

This work was largely supported by NSF under grants ANI–0082493 and CNS–0335296, and by Cisco Systems' generous equipment donations.

References

[1] Dynamic network emulation backplane project. http://www.cc.gatech.edu/computing/pads/nms/.

[2] Limitations in NS. http://www.isi.edu/nsnam/ns/ns-limitations.html.

[3] W. Almesberger. UML Simulator. In *Ottawa Linux Symposium*, 2003.

[4] A. Boukerche and C. Tropper. A static partitioning and mapping algorithm for conservative parallel simulations. In *Proceedings of the Eighth Workshop on Parallel and Distributed Simulation*. ACM Press, 1994.

[5] J. Cowie, H. Liu, J. Liu, D. Nicol, and A. Ogielski. Towards realistic million-node internet simulations. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, Jun. 1999.

[6] J. Cowie, D. Nicol, and A. Ogielski. Modeling the global internet. *Computing in Science and Engineering*, 1(1):42 – 50, 1999.

[7] David M. Nicol. Comparison of Network Simulators Revisited.

http://www.ssfnet.org/Exchange/gallery/dumbbell/dumbbell-performance-May02.pdf.

[8] K. Fall. Network emulation in the vint/ns simulator. In *Proc. of the 4th IEEE Symposium on Computers and Communications*, 1999.

[9] R. M. Fujimoto. Parallel discrete event simulation. In *Communications of the ACM*, volume 33, pages 30–53, October 1990.

[10] S. Guruprasad. Issues in integrated network experimentation using simulation and emulation. Master's thesis, University of Utah, May 2004. Draft. www.cs.utah.edu/flux/-papers/guruprasad-draftthesis-base.html.

[11] M. Hibler, L. Stoller, R. Ricci, J. L. Duerig, S. Guruprasad, T. Stack, and J. Lepreau. Virtualization Techniques for Network Experimentation (under submission). May 2004.

[12] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientifi c Computing*, 20(1):359–392, 1998.

[13] J. Liu and D. Nicol. Learning not to share. In *Proceedings of the fi fteenth workshop on Parallel and distributed simulation*, pages 46–55. IEEE Computer Society, 2001.

[14] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *Symposium on Operating Systems Principles*, pages 217–231, 1999.

[15] M. Neufeld, A. Jain, and D. Grunwald. Nsclick: Bridging network simulation and deployment. In *Proc. of the 5th ACM International Workshop on Modeling, Analysis, and Simulation of Wireless and Mobile Systems (MSWiM '02)*, pages 74–81, Atlanta, GA, Sept. 2002.

[16] R. Ricci, C. Alfeld, and J. Lepreau. A solver for the network testbed mapping problem. *ACM SIGCOMM Computer Communications Review*, 2003.

[17] G. F. Riley, R. Fujimoto, and M. H. Ammar. A generic framework for parallelization of network simulations. In *Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1999.

[18] L. Rizzo. Dummynet and forward error correction. In *Proc. of the 1998 USENIX Annual Technical Conf.*, New Orleans, LA, June 1998. USENIX Association.

[19] J. Touch and S. Hotz. The X-bone. In Third Global Internet Mini-Conference in conjunction with Globecom, Novemeber 1998.

[20] J. Touch, Y. shun Wang, L. Eggert, and G. G. Finn. A virtual internet architecture. Technical Report ISI-TR-2003-570, Information Sciences Institute, 2003.

[21] UML Simulator Project. http://umlsim.sourceforge.net/.

[22] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 271–284, Boston, MA, Dec. 2002.

[23] S. Y. Wang and H. T. Kung. A simple methodology for constructing extensible and high-fi delity tcp/ip network simulators. In *Proc. of the 18th Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM '99)*, volume 3, pages 1134–1143, 1999.

[24] S. Y. Wang and H. T. Kung. A new methodology for easily constructing extensible and high-fi delity tcp/ip network simulators. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 40(2):257–278, 2002.

[25] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002.