NEPI: Using Independent Simulators, Emulators, and Testbeds for Easy Experimentation

Mathieu Lacage mathieu.lacage@inria.fr
Martin Ferrari martin.ferrari@inria.fr
Mads Hansen mads.hansen@inria.fr
Thierry Turletti thierry.turletti@inria.fr
Planete project-team, INRIA, France

Abstract

Evaluating new network protocols, applications, and architectures uses many kinds of experimentation environments: simulators, emulators, testbeds, and sometimes, combinations of these. As the functionality and complexity of these tools increases, mastering and efficiently using each of them is becoming increasingly difficult.

In this paper, we consider how to make it easier to use multiple tools separately and together to improve the productivity of network researchers. We show how a single object model which encompasses every aspect of a typical experimentation workflow can be used to completely describe experiments to be run within very different experimentation environments.

Although NEPI is still in early design and prototyping stage, we expect that its ability to describe and automate easily complex *mixed* experiments will enable further experimentation with heterogenous networks.

1 Introduction

Sparked by a need for solid investigation efforts before deployments in the real world, experimentation tools of every kind have proliferated over the past ten years. Many stem from the fact that the time and space varying characteristics of the Internet as a whole cannot be captured within a single facility such as PlanetLab, Emulab, ORBIT, or ModelNet. The evaluation of a new protocol or a new application requires using multiple experimentation environments: simulators, emulators, highly controlled testbeds, and sometimes complex combinations of these.

Unfortunately, the cost of doing the right thing, that is, use multiple tools to investigate varying experimental conditions, is often unrealistically high. This cost mostly comes from:

- having to learn new programming languages and interfaces, new tools, and new authentication and authorization mechanisms to use each testbed,
- being unable to keep track of all the experimentation details over many months to ensure that experimentation conditions can be accurately reproduced later. This includes a description of the network and application level aspects of an experiment, but also deployment, setup, and, monitoring scripts, and, more generally, everything needed to reproduce the same experiment.

These problems are, of course, amplified when using multiple testbeds at the same time (so-called *mixed* experiments). For example, it is very hard to maintain a good grasp of the overall experiment when its network topology and its application setup descriptions are split among many separate files written using different languages and APIs.

In this paper, we present the preliminary prototype of the *Network Experiment Programming Interface* (NEPI) whose goal is to make it possible for mere mortals to use many different experimentation environments, and switch among them easily. NEPI intends to make it possible to write a single script to control every aspect of a potentially mixed experiment, including a hierarchical network topology description, application-level setup, deployment, monitoring, trace setup, and trace collection. The greatest challenge NEPI attempts to overcome is the highly variable level of detail required to describe an experiment using different experimentation tools. We want NEPI to export all the functionality of every tool but we want to do this through a uniform programming interface to minimize the learning curve for new testbeds.

Since NEPI is still in very early design and prototyping stages, it provides for now a single backend implementation for ns-3 and lacks real usability tests. As we gain experience with more backend implementations and gather feedback from early users, we expect to revise considerably the programming interface presented in this paper. We outline the status of our current prototype in Section 3 and then, describe in Section 4 the testcase we have been using to evaluate it.

2 Related Work

Many other projects have attempted to solve similar problems but none of them have tried to present behind a unified programming interface the description of both the network and application layers of an experiment running on multiple separate testbeds.

Splay [7] and Plush [5] focus solely on the

application-level deployment. Splay requires users to rewrite their applications using the lua programming language and allows them to control their deployment through the same lua scripts while Plush uses XML files to describe which arbitrary applications should be deployed where.

PlanetLab [6] and ModelNet [9] deal only with the network-level description and delegate deployment to the user. ModelNet converts the user-provided XML-based layout of a wired network into an emulated version of the topology and then, lets Splay, Plush, or others take over. Similarly, PlanetLab allows users to specify which nodes are part of a slice through an XML-RPC or a web interface and, then, allows the user to remotely log into the slice.

Others, such as ORBIT [8] and its management software OMF [1], provide tools to describe network topologies as well as manage the deployment of applications using ruby scripts but they do so only within a single testbed. Emulab [10] stands out from every other project in that it provides a limited form of multiple-testbed support together with the ability to describe both the network and the application level aspects of an experiment. The cost of that feature, however, is that every testbed must be integrated within Emulab itself which makes it especially problematic to use new testbeds independently from Emulab.

3 The NEPI Framework

NEPI is a python library whose goal is to provide all the facilities needed to accomplish every task of a typical experimentation workflow across various testbeds:

- describe the network and application level aspects of an experiment,
- enable trace collection at various key locations within an experiment,
- start, monitor, and stop a running experiment, and
- collect the trace results of an experiment once it is completed.

Interaction with NEPI takes place through python scripts or a graphical user interface: those who dislike GUIs or who need to run batches of experiments will most likely focus on writing a script to control NEPI.

3.1 The Architecture

As depicted in Figure 1, the entire API is accessed through a single `Controller` entity. The controller provides access to a potentially remote daemon that is responsible for managing experiments, and allows users to attach and detach from each of them. Because the failure of this daemon (crash or host reboot) would mean losing access to every running experiment monitored by this daemon, it is especially important to be able to start this daemon on a separate stable server-class host rather than on a workstation or laptop for long running experiments. So far, we have assumed that the lifetime of the controller is longer than that of any experiment (which means, potentially, many months of uptime for some scenarios). This is obviously an unrealistic assumption but we expect that dealing gracefully with controller failures will merely require changes to our implementation and will not impact the user-visible architecture.

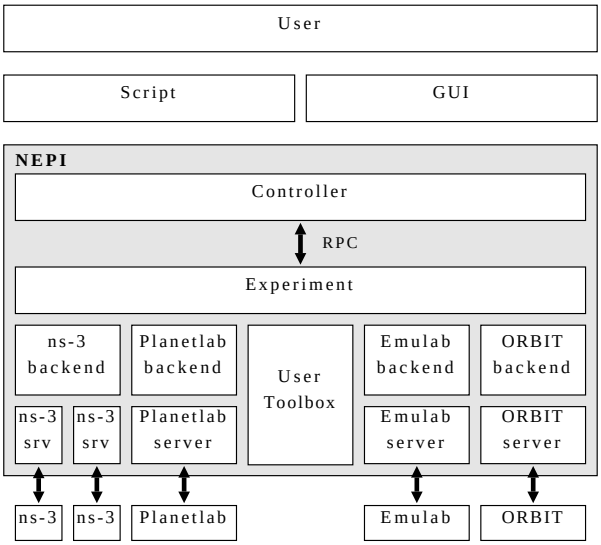

Figure 1: The NEPI architecture

`Experiment` objects hold the complete description of an experiment. They keep track of the list of results, and allow selectively downloading the associated data to a filesystem store. These objects can also save and load the experiment description to and from an XML file. They are responsible for orchestrating the deployment, and the status monitoring of every remote host. By default, remote host failures are handled in a very primitive way: they shutdown the experiment and generate an error message. This simple behavior will allow us to gain insight to the kind of errors which occur most often in typical experimentation scenarios. We then expect to draw on existing tools such as Plush [5] to implement more advanced error management policies and to make them configurable at a later time.

The `Backend` and `Server` objects encapsulate the logic needed to interface with a specific experimentation environment. The server represents a resource such as an ns-3 process or an Emulab *boss* server, running in a possibly remote computer. The backend focuses on providing an implementation of the NEPI object model for a specific experimentation environment. We expect that each backend will export objects which map as closely as possible to the underlying experimentation environ-

ment. The user toolbox will be used to smooth these impedence mismatches across backends: it will collect higher-level aggregate objects for each backend to hide lower-level semantic differences and allow easy switching across them.

3.2 The Object Model

A common way to describe a network topology, but also a complex system made of multiple modules is through a set of boxes interconnected by arrows. Such diagrams are commonly used to design Integrated Circuit board layouts, but also are often the basic blocks of graphical programming languages. This *box metaphor* is very simple but powerful enough to describe arbitrarily-complex systems, especially when composite objects are used to describe hierarchical systems with varying levels of detail.

NEPI uses six kinds of entities to describe an experiment: `Object`, `Connector`, `Attribute`, `Trace`, `Result`, and, `Operation`.

- An `Object` is a box which represents a functional unit of a specific type. An object can represent a functional unit of arbitrary complexity; for example, a simple ns-3 queue or a complete Emulab node. Every functional unit modeled by NEPI is a subtype of this type.
- A `Connector` is a labelled port within an `Object` which can be connected to another connector within another object; for example, the ns-3 `Node` object is connected to the `Ipv4` object through a connector named *protocols* in `Node` and another connector named *node* in `Ipv4`.
- An `Attribute` is the configuration parameter of an `Object` which can be changed both before and while an experiment is running. An example attribute is the throughput of a communication link.
- A `Trace` object represents an event within an `Object` which can be written to a trace file. A trace controls whether or not the event will be enabled, and if yes, it describes in which file this event will be stored.
- A `Result` describes something which has been generated by an `Object` during an experiment; for example, it is typically a pair of strings which indicate on which host and within which file some data can be found.
- An `Operation` can be scheduled to be executed on an `Object` at any point while an experiment is running; for example, an *open-xterm* operation could be used to create a new xterm window connected to a bash shell on a remote virtual machine in ModelNet or CORE [4].

It is possible to define many variations upon this object model. For example, some of our early designs did not include `Connectors`: instead, each connection referenced a pair of `Objects` directly and we inferred the semantics of a connection from the type of the objects connected. Unfortunately, this simple model did not allow us to represent parent/child relationships between objects of the same type. Although we could have used directed connections or connection ordering to infer semantics, we felt that introducing explicitly named connectors would be more user-friendly.

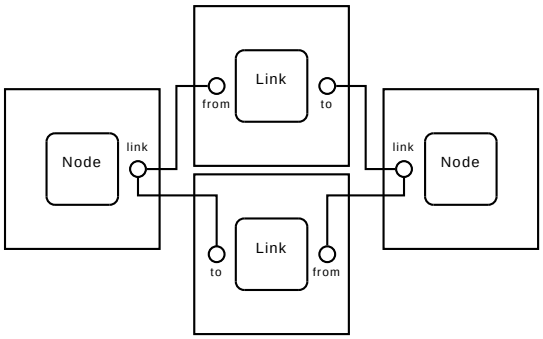

Figure 2: Detail of two ModelNet nodes connected thru unidirectional links

3.3 Applying The Object Model

Because the NEPI object model is very flexible, it is possible to construct many very different representations of the same experiment but choosing the right one is hard. In this section, we outline the representation we expect to use for a few select experimentation environments. We hope to show through these examples the kind of considerations which need to be taken into account to design a backend for an experimentation environment.

The topology of a **ModelNet** [9] experiment is typically described by an XML file which contains a list of core and leaf network nodes as well as a list of unidirectional point to point links between pairs of nodes which are characterized by a delay, throughput, and transmission queue length. Figure 2 shows the simplest way such a network topology can be modeled: two kinds of functional units (`Nodes` and `Links`) and a few attributes for `Link` objects such as delay, throughput and queue length are sufficient to describe the network topology.

A ModelNet backend for NEPI could easily turn the above description into a running instance but application-level traffic would be missing. A ping application is easy to model with an extra functional unit `Ping` interconnected with a `Node` to express which machine it will run on but the destination of the ping needs to be specified: the challenge of modeling applications is in the identification of communication endpoints. Adding a `Destination` attribute to the `Ping` object would require the user to statically assign IP ad-

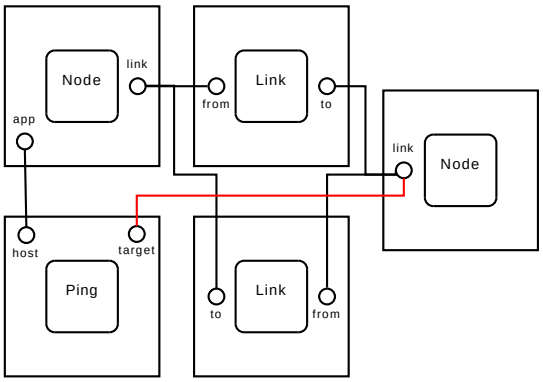

Figure 3: Two connected ModelNet nodes, highlighting the use of connections as references to remote hosts

dresses, hostnames or VNs (ModelNet unique ids) to each leaf node. Figure 3 shows a more graphical option which abstracts the user away from either of these unique ids: the `Ping` application is directly connected to the remote `Node`.

PlanetLab [6] offers very little control over the network topology of an experiment: it merely allows selection of hosts interconnected through the Internet based on their hostname and IP address and provides metadata about each host (geographical location, bandwidth limit, etc.). This, however, is sufficient to allow the NEPI backend of PlanetLab to define a `Location` attribute on a per-`Node` basis to automatically select a host during deployment based on a geographical position. Other attributes to allow selection based on other criteria such as availability or uptime, are also possible if the relevant information is available.

To allow high extensibility, and very high control over its exact behavior, **ns-3** [2] exports natively its functionality through very low-level ns-3 objects: some of these represent layer-3 IP stacks, others represent wireless propagation delay models, etc. From the perspective of an ns-3 user, it thus makes sense to model an ns-3 simulation in NEPI by a set of objects, attributes, and traces which map as much as possible one to one with ns-3 native objects. The greatest challenge to implement this approach is that ns-3 does not use natively such a descriptive approach: a lot of its configuration is functional and is very hard to express in terms of static inter-connection descriptions among objects.

An ns-3 experiment description is shown in Figure 4. It is inherently more low-level than the description used in Figure 3 but experimenters could use NEPI to define a toolbox of higher-level composite objects for conveniance. For example, they could define a `Ns3WirelessNode`, or a `Ns3DumbellNetwork` to make it easier to describe large complex experiments from low-level objects.

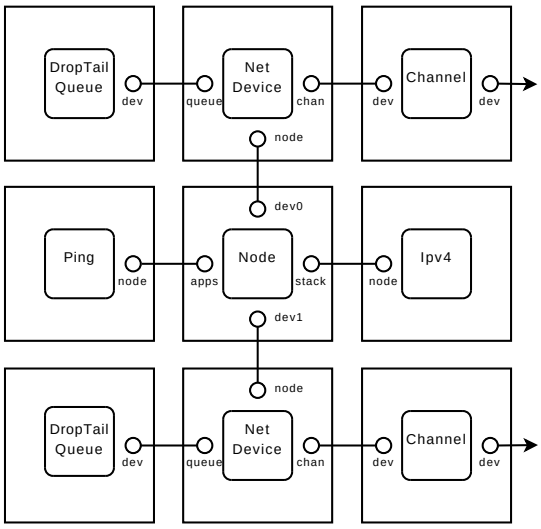

Figure 4: Description of a multihomed host in ns-3

NEPI provides a considerable amount of flexibility to define these new composite objects: users can define simple hierarchical aggregates by describing a new object as a box around a set of inner objects. The connectors, attributes, traces, and results of such an aggregate object are defined as a subset of those found in the inner objects.

Another approach is to directly write a plugin which defines a new object type with a set of adhoc connectors, attributes, traces, and results, and then, to program the behavior of that object by creating and manipulating other NEPI objects. This allows the definition of a `Ns3DumbellNetwork` object with attributes such as `LeftLeafNumber` which are used to parameterize the creation of other objects rather than directly control an attribute within an inner object.

Omnet++ [3], uses an adhoc programming language, NED, to construct simulation topologies. The object model implemented by NED is very close to the NEPI object model which makes it trivial to define a one to one mapping from NED objects to NEPI objects. Small differences such as the lack of directed connections in NEPI are easily overcome by adding *in* and *out* prefixes or postfixes to connector names in NEPI. However, the converse is not true: NED was designed as a declarative language to allow two-way GUI editing (editing both automatically generated and hand-writen NED files from the GUI), which makes it impossible to map the arbitrary programming constructs used to define objects in NEPI to NED objects.

These simple examples outline an important feature of NEPI: each backend is able to define whichever functional units make the most sense to model an experiment. Some backends such as PlanetLab will naturally be modeled with fairly high-level functional units but

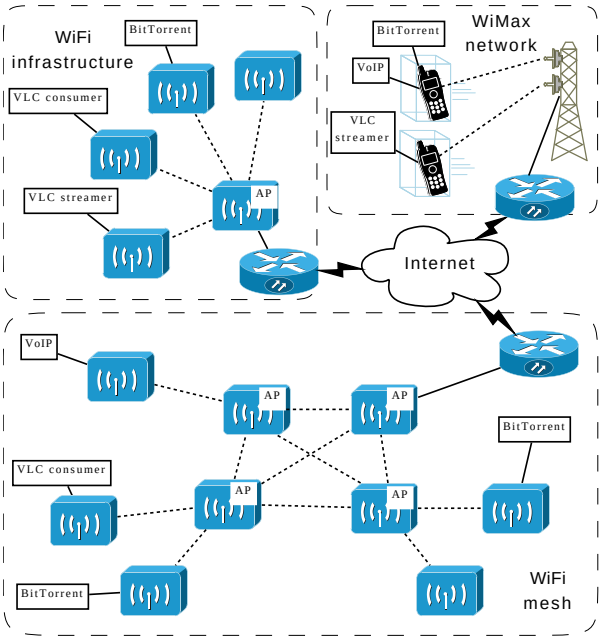

Figure 5: Conceptual diagram of a sample experiment

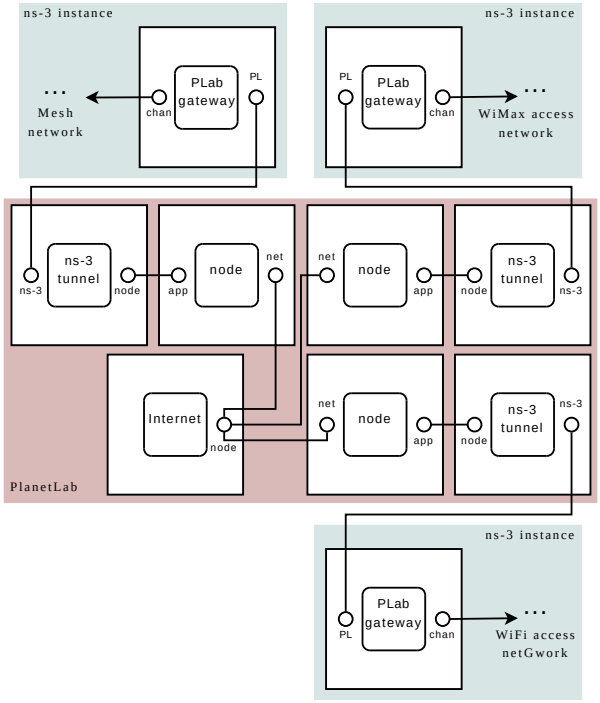

Figure 6: Partial representation of the experiment in NEPI

other backends will more easily map to very low-level functional units. In both cases, NEPI allows the backend to choose whatever is the most convenient but also ensures that the resulting API is coherent with the API exported by every other backend.

4 A Sample Use Case

To illustrate the potential of the NEPI framework, we consider a complex but realistic research experiment. Figure 5 shows a high-level description of this experiment: three access networks (wimax, wifi mesh, and wifi infrastructure) comprising 50 hosts each are interconnected by the Internet. On each leaf host, socket-based applications such as VLC over RTP, BitTorrent peers, or VoIP over SIP generate traffic.

There are many ways to make this experimentation scenario come true. The most obvious solution is to buy some fancy wimax hardware for the wimax base station, invest money in a set of mesh wifi routers, spend more money to setup a reproducible radio environment, etc.

Another much cheaper and much easier to deploy solution is to simulate these wireless networks: ns-3, for example, can be used with wifi and wimax modules to provide highly-reproducible wireless links and work is under way to allow normal socket-based applications to be run within the simulator.

The Internet connections between each access network could also be simulated but it would be hard to ensure realistic background traffic and error conditions on these links. A tempting solution here is to run a separate realtime simulation for each access network on three hosts interconnected by the Internet to ensure that the experiment is exposed to the variable traffic conditions of the real Internet.

Since PlanetLab was specifically designed to allow users to allocate a set of hosts interconnected by the Internet: setting up the above experiment is thus merely a matter of requesting the creation of a PlanetLab slice, assign 3 nodes to this slice, and then, start a realtime ns-3 simulation on each remote host for each access network.

The main challenge to get this scenario running is that, to ensure that each simulated host within each simulator is able to send packets to other simulated hosts within other simulators, we need to allocate at least one testbed-routable IP address per simulated network interface and, then, correctly setup the routing table of each simulated host.

We could manually request the allocation of routable IP addresses within each of the hosting entities of the PlanetLab nodes we assigned to our slice, but that request would be unlikely to be answered positively. We thus have to create our own private IP network, ensure that all addresses are allocated coherently, and tunnel our simulated IP packets over the Internet from one PlanetLab host to another. At this point, it should be pretty clear that, while all of this is perfectly feasible technically, it is far from trivial to do it manually.

The initial approach to automation would naively use a unix shell script and ssh to remotely install ns-3, up-

load ns-3 scenario descriptions, configure a set of IP tunnels, and, finally, start each simulation. This simple solution would still require manually allocating ranges of IP addresses for each access network, hardcoding them in each separate ns-3 simulation script, and providing the right configuration parameters for each IP tunnel. The big missing piece here is that the automation script does not have a global view of the network-level aspects of the experiment. i.e., it cannot assign IP addresses or setup IP tunnels on its own because it has no information about the simulated topology of each access network.

Unsurprisingly, because NEPI was designed to pull together in a single script the description of a complete, potentially *mixed* experiment, it is able to fully automate all the configuration tasks mentionned above.

To describe this experiment with NEPI, we begin by creating four `Server` objects: one for PlanetLab to represent our slice, and the other three for ns-3. In the PlanetLab server, we create three `Nodes` to represent each host allocated to the slice and connect them all to the `Internet` object. The `Ns3Tunnel` objects attached to each node represents the tunneling binary running within each PlanetLab host: it is connected to a `PlabDevice` running within the associated ns-3 simulation. Figure 6 summarizes this description.

When the user starts this experiment, NEPI first adds three hosts to the pre-existing PlanetLab slice, and waits until the relevant sliver is started on each host. Once this is done, it remotely starts the tunneling application and ns-3 on each sliver, and then, remotely configures the IP addresses and routing table of each simulated node as well as the tunnels in each tunneling application.

5 Conclusion

The goal of NEPI is to give researchers more freedom in choosing which testbed to use and to make it possible to automate and run complex mixed experiments. It strives to provide a uniform yet flexible interface to describe and control all aspects of a network experiment.

The open question we are trying to answer is whether or not it is possible to define a uniform object model that can be used with arbitrary network experimental environments such that differences in the underlying attributes or capabilities of the tool can be successfully reconciled.

Because of the richness of the models exported by a network simulator, we have, so far, focused our work on the implementation of a NEPI backend for ns-3 to evaluate how flexible the NEPI object model really is.

We expect to release a working prototype of NEPI during the summer of 2009 with a functional ns-3 backend. Once this is done, we intend to focus on adding backends for other testbeds such as PlanetLab, Emulab, ORBIT, and ModelNet.

Although a lot of work remains to be done to implement multiple backends for NEPI, and to evaluate the usefulness of NEPI in practice, we believe that tools such as these are critical to enabling further experimentation with heterogenous networks.

6 Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n 224263.

References

[1] OMF: `http://omf.mytestbed.net`

[2] ns-3: `http://www.nsnam.org`

[3] Varga, András . "The OMNeT++ Discrete Event Simulation System" *Proc. of the European Simulation Multiconference, 2001.*

[4] Ahrenholz, Danilov, Henderson, and Kim. "CORE: A real-time network emulator" *Proc. of the IEEE Military Communications Conference, 2008.*

[5] Albrecht, Braud, Dao, Topilski, Tuttle, Snoeren, and Vahdat. "Remote Control: Distributed Application Configuration, Management, and Visualization with Plush" *Proc. of the 21st conference on Large Installation System Administration.*

[6] Bavier, Bowman, Chun, Culler, Karlin, Muir, Peterson, Roscoe, Spalink, and Wawrzoniak. "Operating System Support for Planetary-Scale Network Services." *Proc. of the First Symposium on Networked Systems Design and Implementation.*

[7] Leonini, Riviere, and Felber. "SPLAY: Distributed Systems Evaluation Made Simple." *Proc. of the 6th USENIX Symposium on Networked Systems Design and Implementation.*

[8] Raychaudhuri, Seskar, Ott, Ganu, Ramachandran, Kremo, Siracusa, Liu and Singh. "Overview of the ORBIT radio grid testbed for evaluation of next-generation wireless network protocols." *Proc. of the Wireless Communications and Networking Conference*

[9] Vahdat, Yocum, Walsh, Mahadevan, Kostic, Chase, and Becker. "Scalability and Accuracy in a Large-Scale Network Emulator" *Proc. of the 5th Symposium on Operating Systems Design and Implementation.*

[10] White, Lepreau, Stoller, Ricci, Guruprasad, Newbold, Hibler, Barb, Joglekar. "An Integrated Experimental Environment for Distributed Systems and Networks." *Proc. of the 5th Symposium on Operating Systems Design and Implementation.*