

# An Experimentation Workbench for Replayable Networking Research

Eric Eide            Leigh Stoller            Jay Lepreau

*University of Utah, School of Computing*  
{eeide, stoller, lepreau}@cs.utah.edu    www.emulab.net

Flux Technical Note FTN–2006–03  
December 11, 2006

## Abstract

The network and distributed systems research communities have an increasing need for “replayable” research, but our current experimentation resources fall short of reaching this goal. Replayable activities are those that can be re-executed, either as-is or in modified form, yielding new results that can be compared to previously obtained results. Replayability requires complete records of experiment processes and data, of course, but it also requires facilities that allow those processes to actually be examined, repeated, modified, and reused for new studies.

We are now evolving Emulab, our popular network testbed management system, to be the basis of a new “experimentation workbench” in support of realistic, large-scale, replayable networking research. We have implemented a new model of testbed-based experiments, based on scientific workflow, that allows people to move forward and backward through their experimentation processes. Integrated tools in the workbench help researchers manage their activities (both planned and unplanned), software artifacts, data, and analyses. In this paper we present the workbench, describe its implementation, and report how it has been used by early adopters. Our initial case studies, using PlanetLab nodes, cluster PCs, 802.11 wireless, and software radio, have highlighted both the utility of the current workbench and additional usability challenges that must be addressed.

## 1 Introduction

In the networking and operating systems communities, there is an increasing awareness of the benefits of repeated research [6, 18]. A scientific community advances when its experiments are published, subjected to scrutiny by peers, and repeated to determine the veracity of results. Repeated research not only helps to validate the

conclusions of previous studies, but also to expand on those conclusions and suggest new directions for future research.

To repeat a piece of research, one first needs access to the complete records of the experiment that is to be redone. This obviously includes the results of the experiment—not only the final data products, but also the “raw” data products that are the bases for analysis. In the networking community especially, there has been much focus on publishing and archiving the data sets that are the bases of scientific conclusions [3, 4, 7, 28]. Data sets allow researchers to repeat experimental analyses, but by themselves, they do not help researchers validate or repeat the data collection process itself. Therefore, the records of a repeatable experiment must also contain descriptions of the procedures that were followed, in sufficient detail to allow them to be re-executed. For studies of software-based systems, the documentation of an experiment can also contain copies of the actual software that was executed, test scripts, and so on.

A second requirement for repeated research is access to experimental infrastructure: i.e., laboratories. In the networking community, this need is increasingly being served by a variety of network testbeds: environments designed to provide resources for scalable and “real-world” experimentation. Some testbeds (such as Emulab [32]) focus on providing high degrees of control and repeatability, whereas others (such as PlanetLab [25]) focus on exposing networked systems to actual Internet conditions and users. Although modern testbeds differ in many respects, most are similar in that they are primarily designed to provide experimenters with access to resources. Once a person has obtained resources, he or she generally receives little help from the testbed in actually performing an experiment: i.e., configuring it, executing it, and collecting data from it. Current testbeds offer little help to users that want to repeat research, and moreover, they provide little guidance toward making new experiments repeatable.

---

This material is based upon work supported by NSF under grants CNS–0524096, CNS–0335296, and CNS–0205702.

Based on our experience in running and using Emulab, we believe that new *testbed-integrated, user-centered tools* will be a necessary third requirement for establishing repeatable research within the networking community. Emulab is our large and continually growing testbed: it provides uniform access to many hundred computing devices of diverse types, in conjunction with user services such as file storage, file distribution, and user-scheduled events. As Emulab has grown over the past six years, its users have performed increasingly large-scale and sophisticated studies. An essential part of these activities is managing the many parts of every experiment, and we have seen first-hand that this burden can be a heavy load for Emulab’s users. As both administrators and users of our testbed, we recognize that network researchers need better ways to organize, execute, record, and analyze their work.

In this paper we present our evolving solution: an integrated experimentation management system called the *experimentation workbench*. The workbench uses concepts from scientific workflow management systems to provide new ways for Emulab’s users to structure their activities. The workbench is based on a new model of testbed-based experiments, one that is designed to describe the relationships between multiple parts of experiments and their evolution over time. The workbench enhances Emulab’s previous model and existing features in order to help researchers “package” their experiment definitions, explore variations of experiments, capture the inputs and outputs of experiments, and perform data analyses. A key concern of the workbench is automation: we intend for users to be able to re-execute testbed-based experiments with minimum effort, either as-is or in modified form. Repeated research requires both experiment encapsulation and access to a laboratory, but the workbench goes further by automating the execution of experiments. Thus, instead of saying that the workbench supports repeated research, we say that it supports *replayable research*: activities that not only can be re-executed, but that are based on a framework that makes re-execution straightforward.

The primary contributions of this paper are (1) the identification of *replayable research* as a critical part of future advances in networking; (2) the detailed presentation of our *experimentation workbench*, which is our evolving framework for replayable networking research; and (3) an evaluation of the current workbench through *case studies of its use* in actual research projects. Our workbench is implemented atop Emulab, but the idea of replayable research is general and applicable to other testbed substrates. In fact, two of our case studies utilize PlanetLab via the Emulab-PlanetLab portal [31].

This paper builds on our previous work [11] by detailing the actual workbench we have built, both at the con-

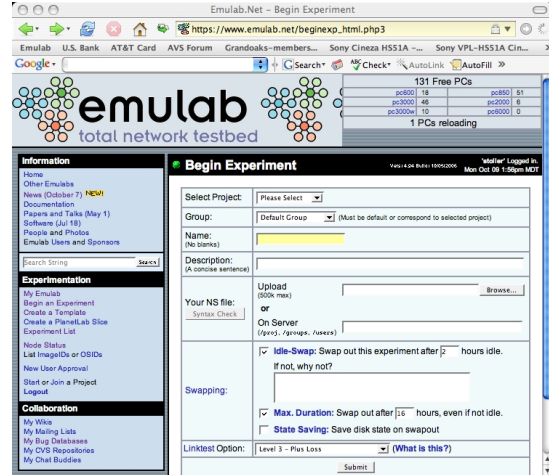


Figure 1: Emulab’s Web interface

ceptual level (Section 3) and the implementation level (Section 4). In addition, our case studies (Section 5) show how the current workbench has been applied to ongoing network research activities, including software development and performance evaluation, within our research group. The case studies highlight both the usefulness of the current workbench and ways in which the current workbench should be improved. Section 6 presents related work, and Section 7 concludes.

## 2 Background

Since April 2000, our research group has continuously developed and operated *Emulab* [32], a highly successful and general-purpose testbed facility and “operating system” for networked and distributed system experimentation. Emulab provides integrated, Web-based access to a wide range of experimental environments including simulated, emulated, and wide-area network resources. It is a central resource in the network research and education communities: as of October 2006, the Utah Emulab site had over 1,500 users from more than 225 institutions around the globe, and these users ran over 18,000 experiments in the preceding 12 months. In addition to our testbed site, Emulab’s software today operates more than a dozen other testbeds around the world.

The primary interface to Emulab is through the Web, as shown in Figure 2. Once a user logs in, he or she can start an *experiment*, which is Emulab’s central operational abstraction. An experiment defines both a static configuration and a dynamic configuration of a network, as outlined in Figure 2. Experiments are typically described using an extended version of the *ns* language [14], but they may also be created through a GUI within Emulab’s Web interface.

```

set ns [new Simulator]
source tb_compat.tcl

# STATIC PART: nodes, networks, and agents.
set cnode [$ns node] # Define a node
set snode [$ns node]
set lan [$ns make-lan "$cnode $snode" \
        100Mb 0ms]
set client [$cnode program-agent]
set server [$snode program-agent]

# DYNAMIC PART: events.
set do-client [$ns event-sequence {
    $client run -command "setup.sh"
    $client run -command "client.sh"
}]
set do-server [$ns event-sequence {
    $server run -command "server.sh"
}]
set do-expt [$ns event-sequence {
    $do-server run
    $do-client run
}]
$ns at 0.0 "$do-expt start"
$ns run

```

Figure 2: A sample experiment definition

The static portion describes a network topology: a set of computers and other devices, the network in which they are contained, and the configurations of those devices and network links. This description includes the type each node (e.g., a 3 GHz PC, a PlanetLab node, or a virtual machine), the operating system and other packages that are to be loaded onto each node, the characteristics of each network link, and so on. It also includes the definitions of *program agents*, which are testbed-managed entities that run programs as part of an experiment.

The dynamic portion is a description of events: activities that are scheduled to occur when the experiment is executed. An event may be scheduled to occur at a particular time, e.g., thirty seconds after the start of the experiment. An event may also be unscheduled: in this case, the user or some running processes may signal the event at run time. Events can be assembled into event sequences as shown in Figure 2. Events are managed and distributed by Emulab and are received by various testbed-managed agents. Some agents are set up automatically by Emulab, including those that receive commands to operate on nodes and links (e.g., to bring them up or down). Other agents are set up by the user as part of an experiment. These include the program agents mentioned above; traffic generator agents, which produce various types of network traffic; and timelines, which are

agents that signal a user-specified sequence of events.

Through Emulab’s Web interface, a user submits an experiment definition and gives it a name. Emulab parses the specification and stores the experiment in its database. The user can now “swap in” the experiment, meaning that it is mapped onto physical resources in the testbed. Nodes and network links are allocated and configured, program agents are created, and so on. When swap in is complete, the user can login to the allocated machines and do his or her work. A central NFS file server provides persistent storage; this server is available to all the machines within an experiment. Some users carry out their experiments “by hand,” whereas others use events to automate and coordinate their activities. When the user is done, he or she tells Emulab to “swap out” the experiment, which releases the experiment’s resources. The experiment itself remains in Emulab’s database so it can be swapped in again later.

### 3 New Model of Experimentation

Over time, as Emulab was used for increasingly complex and large-scale research activities, we realized that the model of experiments described above fails to capture important aspects of the experimentation process.

#### 3.1 Problems

We identified five key ways that the original Emulab model of experiments breaks down for users.

- 1. An experiment entangles the notions of definition and instance.** In other words, an experiment combines the idea of describing a network with the idea of allocating physical resources for that description. This means, for example, that a single experiment description cannot be used to create two concurrent instances of that experiment.

- 2. The old model cannot describe related experiments,** but representing such relationships is important in practice. Because distributed systems have many variables, a careful study requires running multiple, related experiments that cover a parameter space.

- 3. An experiment does not capture the fact that a single “session” may encompass multiple subparts,** such as individual tests or trials. These subparts may or may not be independent of each other: for example, a test activity may depend on the prior execution of a setup activity.

- 4. Data management is not handled as a first-class concern:** i.e., acquisition, collection, organization, and analysis. Users had to deal with instrumenting their systems under study and orchestrating the collection of that data. For a large system with many high-frequency probes, the amount of data gathered can obviously be

very large. Moreover, users need to analyze all their data: not just within one experiment, but across separate experiments.

**5. Finally, the old model does not help users to manage all the parts of an experiment.** In practice, an experiment is not defined just by an *ns* file, but also by all the software, input data, configuration parameters, documentation, and so on that is utilized within the experiment. These things change in both planned and unplanned ways over the course of a study, which may span a long period of time. Saving and recalling history is essential for many purposes including collaboration, reuse, replaying experiments, and reproducing results.

### 3.2 Solution: refine the model

The issues described above central to the experimentation workbench and our vision for replayable research. Addressing these issues, therefore, was an essential first step in the design and implementation of the workbench.

Our solution was to design a new, expanded model of testbed-based experiments. The original model of Emulab experiments is monolithic in the sense that a single user-visible entity represents the entirety of an experiment. This monolithic notion nevertheless fails to capture all the aspects of an experiment as described above. Our new model divided the original notion of an experiment into parts, and then enhances those parts with new capabilities.

Figure 3 summarizes the main relationships between the components of the new model. The two most important components are *templates*, shown at the top, and *records*, shown at the bottom. These two types of components are persistent and are stored “forever” by the experimentation workbench. In contrast, most of the other model components are transient because they represent entities that exist only while testbed resources are being used.

The rest of this section describes the elements of our new model of experimentation at high level. Section 4 shows how these model elements are realized and used in our Emulab-based workbench implementation.

- A **template** is a repository for the many things that collectively define a testbed-based environment. A template plays the “definition” role of Emulab’s original experiment abstraction. A template contains the many files that are needed for a study—not just an *ns* file, but also the source code and/or binaries of the system under test, input files, and so on. In addition, a template may contain other kinds data such as (reified) database tables and references to external persistent storage, e.g., datapositories [4] and CVS repositories.

Templates have two additional important properties. First, templates are *persistent* and *versioned*. A template

is an immutable object: “editing” a template actually creates a new template, and the many revisions of a template form a tree that the user can navigate. Second, templates have *parameters*, akin to the parameters to a function. Parameters allow a template to describe a family of related environments and activities, not just one. A user specifies the values of parameters when he or she instantiates a template, as described next.

- A **template instance** is a container of testbed resources: nodes, links, and so on. A user creates a template instance and populates it with resources by using the workbench’s Web interface (implemented as extensions to Emulab’s Web interface). The process of instantiating a template is very much like the process of swapping in an experiment in the traditional Emulab Web interface. There are two obvious differences, however. First, the user can specify parameter values when a template is instantiated. These parameter values are accessible to processes within the instance and are included in records as described below. Second, a user can instantiate a template even if there is an existing instance that is associated with the template.

A template instance is a dynamic and transient entity: it plays the “resource owner” role of the original, monolithic experiment notion. The resources within a template instance may change over time, as directed by the activities that occur within it. When the activities are finished, the allocated resources are released and the instance is destroyed.

- An **run** is a user-defined context for activities. Conceptually, a run is a container of processes that execute and events that occur within a template instance. In terms of the monolithic Emulab experiment model, the role of a run is to represent a user-defined “unit of work.” In the new model, a user can demarcate separate groups of activities that might occur within a single template instantiation—for example, a user could separate individual trials of a system under test. These separate trials may occur serially or in parallel. (Our current implementation supports only serial runs.) A run is transient, but the events that occur within the run, along with the effects of those events (e.g., output files), are recorded in persistent storage as described below.

Like a template instance, a run can have parameter values, which are specified when a run is created.

- An **activity** is a collection of processes, workflows, scripts, and so on that execute within a run. Having an explicit model component for activities is useful for two reasons. First, it is necessary for tracking the provenance of artifacts (e.g., output files) and presenting that provenance to experimenters in meaningful ways. Second, it is needed for the workbench to manipulate activities in rich ways. For example, the workbench could execute only the portions of a workflow that are relevant to a particular

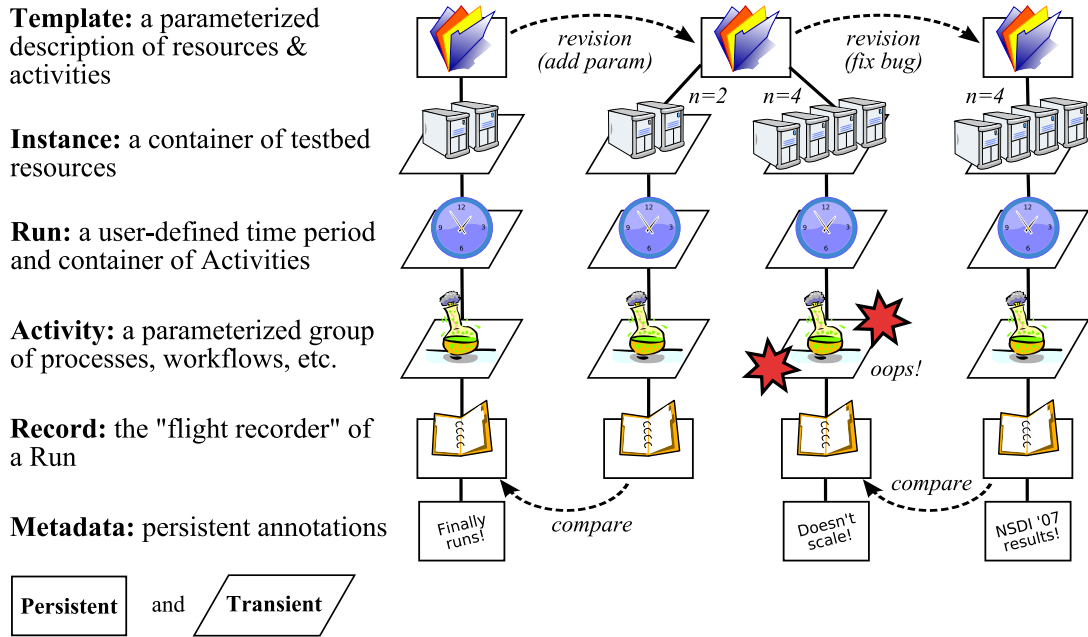


Figure 3: Summary of the new model of experimentation. The diagram illustrates an example of a user’s experimentation over time. (The information is displayed differently in the actual workbench GUI, as shown in other figures.) In the example, the user first creates a template and uses it to get one set of results, represented by the leftmost record. The user then modifies the template by defining a parameter: this is used to create two instances, and the four-node instance fails. The experimenter fixes the bug and runs a final template instance, which succeeds.

output that a researcher wants to generate. In our current implementation, activities correspond to Emulab events and event sequences, plus the actions that are taken by testbed agents when events are received.

- A **record** is the persistent record of the activities and effects that occurred within a run. A record is a repository: it contains output files, of course, but it may also contain input files when those files are not contained within the template that is associated with the record. The idea of a record is to be a “flight recorder,” capturing everything that is relevant to the experimenter and at a user-specified level of detail. Once the record for a run is complete, it is immutable.

The workbench automatically captures data from well-defined sources, and automatic techniques such as packet recorders [12], filesystem monitors [11], and provenance-aware storage systems [21] can help to determine the “extent” of an experiment. Such techniques can be automatically deployed by a testbed and can drastically lower the experiment-specification burden for workbench users.

- Finally, **metadata** is used to annotate templates and records. Metadata are first-class objects, as opposed to being contained within other objects. This is important, because templates and records are immutable once they are created. Metadata can therefore be used to describe other objects, without changing the described ob-

jects themselves. For instance, users can give meaningful names to templates and records through metadata. The workbench itself can also use metadata to attach mutable properties to templates and records—for example, information about how these things should be presented to users in a testbed’s GUI.

## 4 Using the Workbench

In this section we present our implementation of the workbench through an example of its use. Our implementation extends Emulab to support replayable research, based on the conceptual model described in Section 3. Taken as a whole, the model may seem overwhelming to users. An important part of our work, therefore, is to implement the model through GUI extensions and other tools that build upon the interfaces that testbed users are already accustomed to.

### 4.1 Creating templates

A user of the workbench begins by creating a new template. He or she logs in to Emulab Web site and navigates to the form for defining a new template, shown in Figure 4 Readers familiar with Emulab will see that the form is similar to the page for creating regular Emulab “experiments,” except that the controls related to swap-

Figure 4: Creating a new template

ping in an experiment are missing. The form asks the user to specify:

- the *project* and *group* for the template. These attributes relate to Emulab’s security model for users, described elsewhere [32].
- a *template ID*, which is a user-friendly name for the template, and an initial *description* of the template. These are two initial pieces of metadata.
- an *ns* file, which describes a network topology and a set of events, as described in Section 2.

As illustrated later on, the workbench assigns a globally unique identifier (GUID) to each template, and the GUID of a template never changes. In contrast, the user-given template ID is changeable metadata.

**Parameters.** Templates may have parameters, and these are specified through new syntax in the *ns* file. A parameter is defined by a new *ns* command:

```
$ns define-template-parameter name value desc
```

where *name* is the name of the parameter, *value* is the default value, and *desc* is an optional descriptive string. These elements will be presented back to the user when he or she instantiates the template. A parameter defines a variable in the *ns* file, so it may affect the configuration of a template instance. For example, the following code uses a parameter to specify the number of nodes in a network topology:

```
$ns define-template-parameter NodeCount 3
for {set i 1} {$i <= $NodeCount} {incr i} {
  set node($i) [$ns node]
}
```

**Datastore.** At this point, the user can click the *Create Template* button. The *ns* file is parsed, and the template is added to the workbench’s database of templates. Now the user can add other files to the template.<sup>1</sup>

<sup>1</sup>In the future, we will extend the workbench user interface so that users can add files to a template as part of the initial creation step.

The most convenient way for users to add files to a template is by projecting the template into a filesystem. One can think of this as a “checkout” of the template from the repository that is kept by the workbench. Once a template is created, the workbench automatically creates a checkout of the template at a well-known place in Emulab’s file system. The part of the template that contains files is called the *datastore*, and to put files into a template, a user places them within the datastore directory.

```
# Navigate to the datastore of the template.
cd ../datastore
# Add scripts, files, etc. to the template.
cp ~/client.sh ~/server.sh .
cp -r ~/input-files .
```

At this point, the user can “commit” the new files to the template.

```
template_commit
```

In fact, this action creates a *new template*. Recall that templates are immutable, which allows the workbench to keep track of history. Thus, changing the files within a template results in a new template, and the workbench records that the new template is a modification of the original template. The `template_commit` command infers the identity of the original template from the current working directory.

Our current implementation is based on Subversion, the popular open-source configuration management and version control system. This implementation is hidden from users, however: the directories corresponding to a template are not a “live” Subversion sandbox. So far, we have implemented the ability to put ordinary files into a template. Connecting a template to other sources of persistent data, such as databases or external source repositories, is future implementation work. Emulab already integrates CVS (and soon Subversion) support for users, so we expect to connect the workbench to those facilities first.

**Template history.** A template can be modified either through the file system or through Emulab’s Web interface. Each modification results in a new template, and a single template may be modified multiple times—for example, to explore different directions of research. Thus, over a time, a user creates a tree of templates, representing the history of a research study over time. The workbench keeps track of this history and can present it to users as shown in Figure 5. The original template is the root of the tree, at left; the most recent versions of the template are at the right. By clicking on the nodes of the tree, the user can recall and inspect any template in the history. The interface utilizes AJAX to provide a high degree of interactivity. The workbench also provides con-

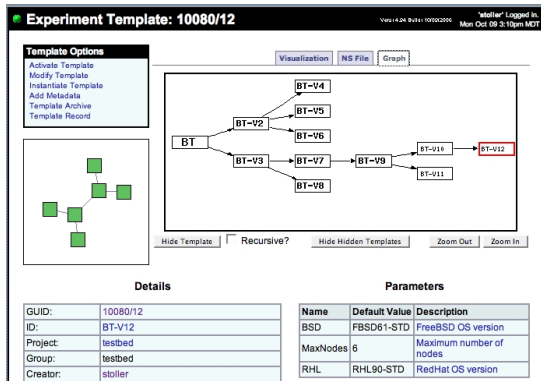


Figure 5: The workbench tracks and displays how templates are derived

ID: (alphanumeric, no blanks)	TemplateTest
Swapping:	<input checked="" type="checkbox"/> <b>Idle-Swap:</b> Swap out this experiment after <input type="text" value="2"/> hours idle. If not, why not? <input type="text"/> <input checked="" type="checkbox"/> <b>Max. Duration:</b> Swap out after <input type="text" value="16"/> hours, even if not idle. <input type="checkbox"/> <b>State Saving:</b> Save disk state on swapout
Formal Parameters:	CLIENT_COUNT <input type="text" value="2"/> DURATION <input type="text" value="60"/> HWTYPE <input type="text" value="pc850"/> or XML file: On Server <input type="text" value="(pxxx), /groups, /users"/>
Use this text area for an (optional) description:	
<input type="text"/>	
<input type="checkbox"/> Batch Mode Instantiation (See Tutorial for more information)	
Linktest Option:	Level 3 - <input type="button" value="(What is this?)"/>
<input type="button" value="Instantiate"/>	

Figure 6: Instantiating a template

trols that affect the tree display: e.g., individual nodes or subtrees can be elided from the display.

## 4.2 Instantiating templates

Through the Web interface, a user can select a template and then instantiate it. The page for instantiating a template is shown in Figure 6. Readers familiar with Emulab will see that the form for instantiating a template is similar to the form for swapping in a regular Emulab experiment, with two primary differences. First, the template form displays the template parameters and allows the user to edit their values. Second, the user must give a name to the template instance; this name is used by Emulab to create DNS records for the machines that are allocated to the template instance. By default, the name of the instance is derived from the name of the template, but the user can edit this value.

When a template is instantiated, a copy of the template’s datastore is created for the instance. This ensures that concurrent instances of a template will not interfere with each other through modifications of the datastore. This also builds on existing Emulab facilities. Users know that Emulab creates a directory for each experiment they run; our implementation extends this by adding template-specific features, such as copies of the datastore, to that directory.

## 4.3 Defining activities

When a template instance is created, testbed resources are allocated, configured, and booted. After the network and devices are up and running, the workbench automatically starts a run—a context for user work, as described previously—and starts any prescheduled activities within that run. The parameters that were specified for the template instance are communicated to the agents within the run.

**Predefined activities.** In our current implementation, activities are implemented using events and agents. Agents are part of the infrastructure provided by Emulab; they respond to events and perform actions such as modifying the characteristics of links or running user-specified programs. We found that the existing agent and event model was well-suited for describing “prescribed” activities within our initial workbench implementation.

As already described, agents and scripted events are specified in a template’s *ns* file, and commonly, events refer to data that is external to the *ns* file. For example, as shown in Figure 2, events for a program agents typically refer to external scripts. The workbench makes it possible to encapsulate these files within templates, by putting them in the template’s datastore. When a template is instantiated, the location of the instance’s copy of the datastore is made available via the `DATASTORE` variable. An experimenter can use that variable to refer to files that are encapsulated within the template, as shown in this example:

```
set do-client [$ns event-sequence {
  $client run -command {$DATASTORE/setup.sh}
  $client run -command {$DATASTORE/client.sh}
}]
```

**Dynamically recording activities.** The workbench also allows a user to record events dynamically, for replay in the future. This feature is important for lowering the barrier to entry for the workbench and supporting multiple modes of use. To record a dynamic event, a user simply executes the `record` command on some host within the template instance. For example, the following records a dynamic event to execute `client.sh`:

```
record client.sh
```

A second instance of `record` adds a second event to the recording, and so on. The workbench provides additional commands allow a user to stop and restart time with respect to the dynamic record, so that the recording does not contain large pauses when it is later replayed. When a recording is complete, it can be edited using a simple Web-based editor. We discuss replay below.

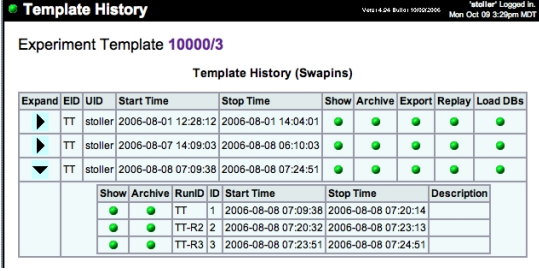
#### 4.4 Using records

In the new model of experimentation, a record is the flight recorder of all the activities and effects that occur during the lifetime of a run. To achieve this goal in a transparent way, the workbench needs to fully instrument the resources and agents that constitute a template instance. We are gradually adding such instrumentation to the testbed, and in the meantime, we use a combination of automatic and manual (user-directed) techniques to decide what should be placed in the permanent record of a run.

**Creating records.** When a run is complete—e.g., because the experimenter uses the Web interface to terminate the template instance—the workbench creates a record of the run containing the following things:

- *the parameter values* that were passed from the template instance to the run. These are stored in an XML file.
- *the logs* that were generated by testbed agents. These are written to well-known places, and so the workbench can collect them automatically.
- *files* that were written to a special archive directory. Similar to the `datastore` directory described previously, every template instance also has a dedicated archive directory in which users can place files that should be persisted.
- *the recorded dynamic events*, as explained previously.
- *a dump of the database* for the template instance. Similar to the archive directory, the workbench automatically creates an online database as part of every template instance. The activities within a template instance can use this database as they see fit. At the end of a run, the contents of the database, if any, are exported and included in the record.

As described for templates, records are also stored in a Subversion repository that is internal to the workbench. This design was chosen as a way to save storage space, since it was anticipated that the different records derived from a single template would be largely similar. Experience has shown, however, that Subversion is too slow for our needs when it is asked to merge large data sets. In response, we have enhanced the production-quality Linux “LVM” logical volume manager software [16].



The screenshot shows a web interface titled "Template History" for "Experiment Template 10000/3". It displays a table of template instances (TT) with columns for Expand, EID, UID, Start Time, Stop Time, Show, Archive, Export, Replay, and Load DBs. Below this, there is a detailed view of runs for a specific template instance, showing columns for Show, Archive, RunID, ID, Start Time, Stop Time, and Description.

Expand	EID	UID	Start Time	Stop Time	Show	Archive	Export	Replay	Load DBs
▶	TT	stoller	2006-08-01 12:28:12	2006-08-01 14:04:01	●	●	●	●	●
▶	TT	stoller	2006-08-07 14:09:03	2006-08-08 06:10:03	●	●	●	●	●
▼	TT	stoller	2006-08-08 07:09:38	2006-08-08 07:24:51	●	●	●	●	●

Show	Archive	RunID	ID	Start Time	Stop Time	Description
●	●	TT	1	2006-08-08 07:09:38	2006-08-08 07:20:14	
●	●	TT-R2	2	2006-08-08 07:20:32	2006-08-08 07:23:13	
●	●	TT-R3	3	2006-08-08 07:23:51	2006-08-08 07:24:51	

Figure 7: Viewing the records associated with a template

LVM already supported filesystem snapshots, but multiple snapshots caused snapshotting to become very slow, due to the way its COW design wrote modified blocks to all snapshots. Our first improvement was to speed up the performance of chained snapshots by an order of magnitude. Our second was to extend LVM to support arbitrary branching of snapshots. Such branching filesystems have until now only been partially implemented in research artifacts [1, 26]. Our implementation is currently working, and when it is production-quality and connected to the workbench, it should solve our performance problem. Perhaps more importantly, it will remove the need for users to put all files that they want saved into the special archive directory tree. However, this “universal” versioning and restoration will raise issues of user intent and user interface, as sometimes they won’t wish certain files, e.g., “.dot” files, to roll back.

**Inspecting records.** The workbench stores records automatically and makes them available to users through the Web interface. From the Web page that describes a template, the user can click on the *Template Record* to view the records that have been derived from that template. Figure 7 shows an example listing of records, which are arranged in a two level hierarchy corresponding to template instances and runs within those instances.

Through the Web interface, a user can navigate to any record and inspect its contents. Additional controls allow a user to reconstitute the database that was dumped to a record: this is useful for post-mortem analysis, either by hand or in the context of another run. Finally, the workbench also provides a command-line program for exporting the entire contents of a record.

**Replay from records.** The Web interface in Figure 7 also includes a *Replay* button, which creates a new template instance from a record. The replay button allows a user to create an instance using the parameters and datastore contents that were used in the original run, whenever that was, since these things are all archived away. The original `ns` file and datastore are retrieved from the template and/or record, and the parameter values come from the record. A new instance is created and replayed, eventually producing a new record of its own.



## 4.5 Managing runs

A user may choose to enclose all of his or her activities within a single run. Alternatively, a user may start and stop multiple runs during the lifetime of a template instance, thus yielding multiple records for the individual activities. An interesting feature of our workbench is that, whenever a new run is started, the user has the opportunity to specify new values of the parameters for that run. The set of parameters are those defined by the template; the user simply has the opportunity to change their values for the new run.

When a run is stopped, the workbench stops all the program agents and tracing agents within the run; collects the logfiles from the agents; dumps the template instance's database; and commits the contents of the instance's archive directory to the record. When a new run is started, the workbench optionally cleans the agent logs; optionally resets the instance's database; communicates new parameter values to the program agents; restarts the program agents; and restarts "event time" so that the scheduled events in the template's *ns* file will re-occur.

An additional capability of the workbench is that users are able to script their entire sequence of runs using the event system and/or Emulab's XML-RPC interface. Using a command line program provided by Emulab, or one written by the user in any language that supports the XML-RPC interface, the user can stop and start new runs, providing new parameter values via an attached XML file. Further, the user can employ either static or dynamic events to launch new runs, or even to launch a script that will connect to the Emulab XML-RPC server.

## 5 Case Studies

Although our experimentation workbench is a work in progress, we wanted to obtain early feedback on the model and implementation from Emulab users. We therefore recruited several members of our own research group to use our prototype. All were experienced testbed users (and developers), but not directly involved in the design and implementation of the workbench. They used the workbench for about a month for their own research in networked systems, as described below.

### 5.1 Study 1: system development

The first study applied the workbench to software development tasks within the Flexlab [10] project. Flexlab is software to support the emulation of "real world" network conditions within Emulab. Specifically, Flexlab emulates conditions observed in PlanetLab, an Internet-based overlay testbed, within Emulab. The goal is to

make it possible for applications that are run within Emulab to be subject to the network conditions that would be present if those same applications had been run on PlanetLab. The Flexlab developers used the workbench to improve the way in which they were developing and testing Flexlab itself.

A Flexlab configuration consists of several pairs of nodes, where each pair contains one Emulab node and a "proxy" of that node in PlanetLab. The Flexlab infrastructure continually sends traffic between the PlanetLab nodes. It observes the resulting behavior, produces a model of the network conditions, and then directs Emulab to condition its network links to match the model. The details of these processes are complex and described elsewhere [10].

The original framework for a Flexlab experiment consisted of (1) an *ns* file, containing variables that control the topology plus the specifics of a particular experiment, and (2) a set of scripts for launching the Flexlab services, monitoring application behavior, and collecting results. An Flexlab developer would start a typical experiment by modifying the *ns* file variables as needed, creating an Emulab experiment with the modified *ns* file, and running a "start experiment" script to start up the Flexlab infrastructure. Thus, although all the files were managed via CVS, each experiment required by-hand modification to the *ns* file, and the Emulab experiment itself was not associated with the files in the developer's CVS sandbox.

At this point the experimenter would run additional scripts to launch an application atop Flexlab. When the application ended, he or she would run a "stop experiment" script to tear down the Flexlab infrastructure and collect results. He or she would then analyse the results, and possibly repeat the start, stop, and analysis phases a number of times. The attributes of each "run" could be changed either by specifying options to the scripts or, in some cases, by modifying the experiment (via Emulab's Web interface). Thus, although the collection of results was automated, the documentation and long-term archiving of these results were performed by hand—or not at all. In addition, the configuration of each run was accomplished through script parameters, not through a testbed-monitored mechanism. Finally, if the user modified the experiment, the change was destructive: going back to a previous configuration meant undoing changes by hand or starting over.

The Flexlab developers used the prototype workbench to start addressing the problems described above with their ad hoc Flexlab testing framework. They created a template from their existing *ns* file, and it was straightforward for them to change the internal variables of their original *ns* file to be parameters of the new template. They moved the start- and stop-experiment scripts into the template datastore, thus making them part of the tem-

plate. The options passed to these scripts also became template parameters: this made it possible for the workbench to automatically record their values, and for experimenters to change those values at the start of each run (Section 4.5). Once the scripts and their options were elements of the template, it became possible to modify the template so that the start- and stop-experiment scripts would be automatically triggered at run boundaries. The Flexlab developers also integrated the functions of assorted other maintenance scripts with the start-run and stop-run hooks. A final but important and immediate benefit of the conversion was that the workbench now performs the collection and archiving of result files from each Flexlab run.

These changes provided an immediate logistical benefit to the Flexlab developers. A typical experiment now consists of starting with the Flexlab template, setting the values for the basic parameters (e.g., number of nodes in the emulated topology, and whether or not PlanetLab nodes will be needed), instantiating the template, and then performing a series of runs via menu options in the workbench Web interface. Each run can be parameterized separately and given a name and description to identify results.

At the end of the case study, the Flexlab developers made four main comments about the workbench and their overall experience. First, they said that although the experience had yielded a benefit in the end, the initial fragility of the prototype workbench sometimes made it more painful and time-consuming to use than their “old” system. We fixed implementation bugs as they were reported, but still they frustrated the Flexlab developers along the way. Second, the developers noted that a great deal of structuring had already been done in the old Flexlab environment that mirrors some of what the workbench provides. Although this can be seen as reinvention, we see it as a validation that the facilities of the workbench are needed for serious development efforts. As a result of the case study to date, the Flexlab developers can now use the generic facilities provided by the workbench instead of maintaining their own ad hoc solutions. Third, the Flexlab developers noted that they are not yet taking advantage of other facilities that the workbench offers. For example, they are not recording application data into the per-instance database (which would support SQL-based analysis tools), nor are they using the ability to replay runs using data from previous records. Fourth, the developers observed that the prototype workbench could not cooperate with their existing source control system, CVS. Integrating with such facilities is part of our design (Section 3.2), but is not yet implemented.

Despite the current limits of the workbench, the Flexlab developers were able to use it to perform real tasks. For instance, the evaluation of Flexlab that ap-

peared in a recent conference submission [9] was carried out with the workbench.

## 5.2 Study 2: performance analysis

We asked one of the Flexlab developers to use the workbench in a second case study, to compare the behavior of BitTorrent on Flexlab to the behavior of BitTorrent on PlanetLab.

He created templates to run BitTorrent configurations on both testbeds. His templates automated the process of preparing the network (e.g., distributing the BitTorrent software), running BitTorrent, producing a consolidated report from numerous log files, and creating graphs using `gnuplot`. Data were collected to the database that is set up by the workbench for each template instance, and the generated reports and graphs were placed into the record. These output files were then made available via Emulab’s Web interface (as part of the record).

Once a run was over, to analyze collected data in depth, the developer used the workbench Web interface to reactivate the live database that was produced during the run. The performance results that the researcher obtained through workbench-based experiments—too lengthy to present here—were included in the previously cited conference submission about Flexlab [9].

In terms of evaluating the workbench itself, the developer in this case study put the most stress on our prototype—and thereby illuminated important issues for future workbench improvements. For example, he initially had problems using the workbench in conjunction with PlanetLab, because our prototype workbench was not prepared to handle cases in which nodes become unavailable between runs. He also asked for new features, such as the ability to change the *ns* file during a template instance, and have those changes become “inputs” to subsequent runs. We had not previously considered the idea that a user might want to change not just parameter values, but the entire *ns* file, between runs in a single template instance. We quickly implemented support for these features, but we will clearly need to revisit the issues that were raised during this case study. In particular, it is clear that the issue of handling testbed resource failures will be important for making the workbench robust and applicable to testbeds such as PlanetLab.

## 5.3 Study 3: application monitoring

In this section, we describe how another local researcher used our workbench to study the behavior of an existing large-scale emulation scenario called GHETE (Giant HETerogeneous Experiment).

GHETE was written by this researcher in June 2006 to showcase Emulab’s ability to handle large network

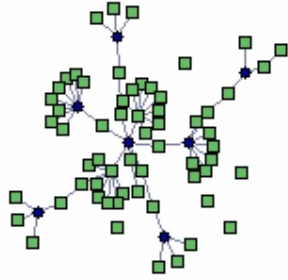


Figure 8: An example GHETE topology. Squares represent nodes, circles represent LANs, and lines represent wired links. Nodes without wired connections have wireless links only. Some nodes act as endpoints and others act as routers.

topologies containing many types of nodes and links: wired and wireless PCs, virtual nodes, software-defined radio nodes, sensor network nodes, and mobile robots carrying nodes. GHETE uses these node types inside an emulation of a distributed data center. The data center consists of two or more remote clusters, each acting as a service center for large numbers of client nodes. Client nodes send large TCP streams of data to the cluster servers via *iperf*, emulating an Internet service, such as a heavily used, distributed file backup service. Each of the clusters is monitored for excessive heat by a collection of fixed and mobile sensor nodes. When excessive heat is detected at a cluster, the cluster services are stopped, emulating a sudden shutdown. A load-balancing program monitors cluster bandwidth and routes new clients to the least-utilized cluster. Figure 8 shows an example GHETE topology. Two service clusters are shown near the top, and several client LANs are shown clustered around the central router.

The GHETE developer used the workbench template system to parameterize many aspects of the emulation, including client network size, bandwidth and latency characteristics, and node operating systems. Because the arguments controlling program execution were also turned into template parameters, the workbench version of GHETE allows an experimenter to execute multiple runs (Section 4.5) to quickly evaluate a variety of software configurations on a single emulated topology.

Although the workbench version of GHETE provided many avenues for exploration, we asked the developer to focus his analysis on the behavior of GHETE’s load balancer. To perform this study, he defined a GHETE topology with two service clusters. The aggregate bandwidth of each cluster was measured at 1 second intervals, and these measurements were stored in the database that is created by the workbench for the template instance. To visualize the effectiveness of the load balancer, the developer wrote a script for the R statistics system [27] that an-

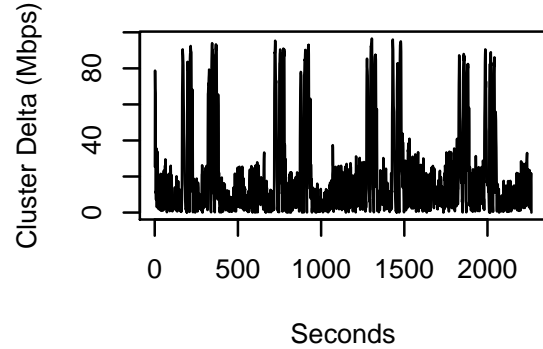


Figure 9: Measured difference in the incoming bandwidths of the two service clusters, using a simple load balancer

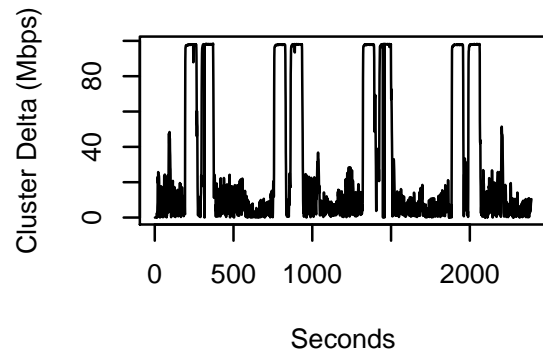


Figure 10: Measured difference in the incoming bandwidths of the two service clusters, using an improved load balancer

alyzes the collected bandwidth measurements. The script automates post-processing of the data: it executes SQL queries to process the measurements directly from the database tables and generates plots of the results.

Figure 9 and Figure 10 show two of the graphs that the developer produced with his R script during the study. Each shows the absolute difference between the aggregate incoming bandwidths of the two service clusters. The spikes in the graphs correspond to times when one cluster was “shut down” due to simulated heat events.

Figure 9 illustrates the behavior of a very simple load-balancing strategy, which exhibits widely varying differences between the two clusters. As long as one cluster’s incoming bandwidth utilization is below the other, clients are routed to the least-utilized cluster. This method produces large oscillations of incoming bandwidth between the two clusters. This occurs because, when the clusters have equalized, the newly added *iperf* clients are still increasing their transmission speed, and the originally least-utilized cluster quickly becomes the most-utilized.

Figure 10 shows the behavior of an improved load balancer that routes clients in a round-robin fashion if the clusters’ incoming bandwidths differ by less than a small threshold. If the bandwidth difference exceeds this

threshold, the load balancer routes one client to the most-utilized cluster for every two clients sent to the least-utilized cluster. This has the effect of reducing oscillation and keeping the clusters slightly more equal in terms of incoming bandwidth.

The GHETE developer said that the workbench was extremely useful in evaluating the load balancing in GHETE and provided insights for future improvements. He listed three primary ways in which the workbench improved upon his previous development and testing methods. First, as noted above, by providing the opportunity to set new template parameter values for each run within a template instance, software controlled by Emulab program agents could be easily configured and reconfigured. This enabled rapid trials and reduced experimenter wait time. Second, via the per-template-instance database that is saved after each run, the GHETE developer was able to analyze incoming data in real time in an interactive R session. He could also easily re-instantiate the database from any previous run for further analysis. Third, since he parameterized all relevant aspects of GHETE’s emulation software, he was able to quickly iterate through many different parameters without wasting time tracking which runs correspond to which parameter settings. Since the parameter bindings for each run are archived by the workbench, he was able to easily recall settings in the analysis phase.

## 6 Related Work

The experimentation workbench is an experiment management system, and there is a growing awareness of the need for such systems in the networking research community. Plush [2], for instance, is a framework for managing experiments in PlanetLab. To use Plush, a user writes an XML file that describes the software that is to be run, the testbed resources that must be acquired, how the software is to be deployed onto testbed nodes, and how the running system is to be monitored. At run time, Plush provides a shell-like interface that helps a user perform resource acquisition and application control actions across many testbed nodes. Plush thus provides features that are similar to those provided by Emulab’s core management services, which utilizes extended *ns* files and provides a Web-based user interface. Our experimentation builds atop these services to address higher-level concerns of experimentation management: encapsulation and parameterization via templates, revision tracking and navigation, data archiving, data analysis, and user annotation. Plush and our work are therefore complementary, and it is conceivable that a future version of the workbench could manage the concerns mentioned above for Plush-driven experiments.

Weevil [30] is a second experiment management sys-

tem that has been applied to PlanetLab-based research. Weevil is similar to Plush and Emulab’s control system in that it deals with deploying and executing distributed applications on testbeds. It is also similar to the workbench in that it deals with parameterization and data collection concerns. Weevil is novel, however, in two primary ways. First, Weevil uses generative techniques to produce both testbed-level artifacts (e.g., topologies) and application-level artifacts (e.g., scripts) from a set of configuration values. Our workbench uses parameters to configure network topologies in a direct way, and it makes parameters available to running applications via program agents. It does not, however, use parameters to generate artifacts like application configuration files, although users can automate such tasks for themselves via program agents. Weevil’s second novel feature is that it places a strong focus on workload generation as part of an experiment configuration. Both of the features described above would be excellent additions to future versions of the workbench. As with Plush, Weevil and the workbench are largely complementary because they address different concerns of replayable research.

Software testing and quality assurance (QA) tasks are common types of experiments, and frameworks for automated quality assurance have much in common with our workbench. Skoll [19], for example, utilizes a distributed testbed to perform QA tasks automatically and in parallel. Skoll provides a process execution framework and features for collecting test results, but more interestingly, it implements novel strategies for exploring the software configuration space, i.e., finding interesting experiments to run. We intend to incorporate similar intelligent steering capabilities in future versions of the workbench. Other popular tools for “automated continuous integration” include CruiseControl [8], Dart [15], and Tinderbox [20]. Although these tools are well-suited to the domain of automated testing within a software development, our experimentation workbench is designed to support many modes of use and a broad range of experiment goals. In addition, the workbench is designed to manage experiments involving distributed systems, which is not a common focus of continuous integration tools.

Many scientific workflow management systems have been developed for computational science, including Kepler [17], Taverna [23], Triana [29], VisTrails [5], and others [33]. Many of these are designed for executing distributed tasks in the Grid. Our workbench has much in common with these systems in that the primary benefits of workflow management include task definition and annotation, tracking data products, and promoting exploration and automation. Our workbench differs from general scientific workflow management systems, however, in terms of its intended modes of use and focus on

networking research. Our experience is that Emulab is most successful when it does not require special actions from its users; the workbench is therefore designed to enhance the use of existing testbeds, not to define a new environment. Moreover, because the workbench is integrated with a network testbed's user interface, resource allocators, and automation facilities, it can do "better" than Grid workflow systems for experiments in networking.

Many experiment management systems have also been developed for domains outside of computer science. For instance, ZOO [13] is a generic management environment that is designed to be customized for research in fields such as soil science and biochemistry. ZOO is designed to run simulators of physical processes and focuses primarily on data management and exploration. Another experiment management system is LabVIEW [22], a popular commercial product that interfaces with many scientific instruments. Our experimentation workbench is a significant step toward bringing the benefits of experiment management, which are well-known in the hard sciences, to the domain of computer science in general and networked and distributed systems research in particular. Networking research presents new challenges for experiment management: for instance, the "instruments" in a network testbed consume and produce many complex types of data including software, input and output files, and databases of results from previous experiments. Networking also presents new opportunities, such as the power of testbeds with integrated experiment management systems to reproduce experiments "exactly" and perform new experiments automatically. Thus, whereas physical scientists must be satisfied with repeatable research, we believe that the goal of computer scientists should be *replayable research*: encapsulated activities *plus* experiment management systems that help people re-execute those activities with minimum effort.

## 7 Conclusion

Vern Paxson described the problems faced by someone who needs to reproduce his or her own network experiment after a prolonged break [24]: "It is at this point—we know personally from repeated, painful experience—that trouble can begin, because the reality is that for a complex measurement study, the researcher will often discover that they cannot reproduce the original findings precisely! The main reason this happens is that the researcher has now lost the rich mental context they developed during the earlier intense data-analysis period." Our goal in building an experimentation workbench for replayable research is to help researchers overcome such barriers—not just for re-examining their own work, but for building on the work of others.

In this paper we have forwarded the idea of *replayable research* for the networking community, which pairs repeatable experiments with the testbed facilities that are needed to repeat and modify those experiments in practice. We have presented the design and implementation of our experimentation workbench that supports replayable research, and we have described how the evolving workbench is being applied by early adopters to actual networking research. Our new model of testbed-based experiments is applicable to network testbeds in general; our implementation extends the Emulab testbed with new capabilities for experiment management. The workbench incorporates and helps to automate the community practices that Paxson suggests [24]: e.g., strong data management, version control, encompassing "laboratory notebooks," and the publication of measurement data. Our goal is to unite these practices with the testbed facilities that are required to actually replay and extend experiments, and thereby advance *science* within the networking and distributed systems communities.

## Acknowledgments

We gratefully thank Kevin Atkinson, Russ Fish, Sachin Goyal, Mike Hibler, and David Johnson for their participation in the case studies. Tim Stack did much early implementation for the workbench, and several other members of our group provided feedback on its current design. Juliana Freire, Mike Hibler, and Robert Ricci all made significant contributions to the workbench design and user interface. Finally, we thank David Andersen for his work on the datapository and Prashanth Radhakrishnan for his work on branching LVM.

## References

- [1] M. K. Aguilera, S. Spence, and A. Veitch. Olive: Distributed point-in-time branching storage for real systems. In *Proc. of the 3rd Symp. on Networked Systems Design and Impl.*, pages 367–380, San Jose, CA, May 2006.
- [2] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. PlanetLab application management using Plush. *ACM SIGOPS Operating Systems Review*, 40(1):33–40, Jan. 2006.
- [3] M. Allman, E. Blanton, and W. Eddy. A scalable system for sharing Internet measurements. In *Proc. of the Passive and Active Measurement Workshop*, Fort Collins, CO, Mar. 2002.
- [4] D. G. Andersen and N. Feamster. Challenges and opportunities in Internet data mining. Technical Report CMU-PDL-06-102, Carnegie Mellon University Parallel Data Laboratory, Jan. 2006. [www.datapository.net](http://www.datapository.net).
- [5] L. Bavoli, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo. VisTrails: Enabling interactive multiple-view visualizations. In *Proc.*

- IEEE Visualization 2005*, pages 135–142, Minneapolis, MN, Oct. 2005.
- [6] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. N. Matthews. Xen and the art of repeated research. In *Proc. of the FREENIX Track: 2004 USENIX Annual Technical Conf.*, pages 135–144, Boston, MA, June–July 2004.
- [7] Cooperative Association for Internet Data Analysis (CAIDA). DatCat. <http://www.datcat.org/>.
- [8] CruiseControl. <http://cruisecontrol.sourceforge.net/>.
- [9] J. Duerig, R. Ricci, D. Gebhardt, M. Hibler, J. Zhang, S. Kasera, and J. Lepreau. Combining the strengths of overlay and emulation network testbeds. Submitted for publication, Oct. 2006.
- [10] J. Duerig, R. Ricci, J. Zhang, D. Gebhardt, S. Kasera, and J. Lepreau. Flexlab: A realistic, controlled, and friendly environment for evaluating networked systems. In *Record of the 5th Workshop on Hot Topics in Networks: HotNets V*, pages 103–108, Irvine, CA, Nov. 2006.
- [11] E. Eide, L. Stoller, T. Stack, J. Freire, and J. Lepreau. Integrated scientific workflow management for the Emulab network testbed. In *Proc. of the 2006 USENIX Annual Technical Conf.*, pages 363–368, Boston, MA, May–June 2006. Short paper.
- [12] Flux Research Group. Emulab tutorial: Link tracing and monitoring. <http://www.emulab.net/tutorial/docwrapper.php3?docname=advanced.html#Tracing>.
- [13] Y. E. Ioannidis, M. Livny, S. Gupta, and N. Ponnkanti. ZOO: A desktop experiment management environment. In *Proc. of the 22th Intl. Conf. on Very Large Data Bases*, pages 274–285, Bombay, India, Sept. 1996.
- [14] ISI, University of Southern California. The network simulator - ns-2. <http://www.isi.edu/nsnam/ns/>.
- [15] Kitware, Inc. Dart: Tests, reports, and dashboards. <http://public.kitware.com/Dart/HTML/Index.shtml>.
- [16] A. J. Lewis. LVM HOWTO. <http://www.tldp.org/HOWTO/LVM-HOWTO>.
- [17] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, Dec. 2005. Accepted for publication.
- [18] J. N. Matthews. The case for repeated research in operating systems. *ACM SIGOPS Operating Systems Review*, 38(2):5–7, Apr. 2004.
- [19] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *Proc. of the 26th Intl. Conf. on Software Engineering*, pages 459–468, Edinburgh, Scotland, May 2004.
- [20] Mozilla Foundation. Tinderbox. <http://www.mozilla.org/tinderbox.html>.
- [21] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proc. of the 2006 USENIX Annual Technical Conf.*, pages 43–56, Boston, MA, June 2006.
- [22] National Instruments. LabVIEW home page. <http://www.ni.com/labview/>.
- [23] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, Dec. 2005. Accepted for publication.
- [24] V. Paxson. Strategies for sound Internet measurement. In *Proc. of the 4th ACM SIGCOMM Conference on Internet Measurement*, pages 263–271, Taormina, Sicily, Italy, Oct. 2004.
- [25] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. *ACM SIGCOMM Computer Communication Review (Proceedings of HotNets-I)*, 33(1):59–64, Jan. 2003.
- [26] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proc. of the 3rd Symp. on Networked Systems Design and Impl.*, pages 353–366, San Jose, CA, May 2006.
- [27] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. <http://www.r-project.org/>.
- [28] C. Shannon, D. Moore, K. Keys, M. Fomenkov, B. Huffaker, and k. claffy. The Internet Measurement Data Catalog. *ACM SIGCOMM Computer Communication Review*, 35(5):97–100, Oct. 2005.
- [29] I. Taylor, I. Wang, M. Shields, and S. Majithia. Distributed computing with Triana on the Grid. *Concurrency and Computation: Practice and Experience*, 17(9):1197–1214, Aug. 2005.
- [30] Y. Wang, M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Automating experimentation on distributed testbeds. In *Proc. of the 20th IEEE/ACM International Conf. on Automated Software Engineering*, pages 164–173, Long Beach, CA, Nov. 2005.
- [31] K. Webb, M. Hibler, R. Ricci, A. Clements, and J. Lepreau. Implementing the Emulab-PlanetLab portal: Experience and lessons learned. In *Proc. of the First Workshop on Real, Large Distributed Systems*, San Francisco, CA, Dec. 2004.
- [32] 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 5th Symp. on Operating Systems Design and Impl.*, pages 255–270, Boston, MA, Dec. 2002.
- [33] J. Yu and R. Buyya. A taxonomy of workflow management systems for Grid computing. Technical Report GRIDS-TR-2005-1, Grid Computing and Distributed Systems Laboratory, Univ. of Melbourne, Mar. 2005.