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

Abstract

The networked and distributed systems research communities have an increasing need for "replayable" research, but our current experimentation resources fall short of satisfying this need. Replayable activities are those that can be re-executed, either as-is or in modified form, yielding new results that can be compared to previous ones. 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.

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 research. We have implemented a new model of testbed-based experiments that allows people to move forward and backward through their experimentation processes. Integrated tools help researchers manage their activities (both planned and unplanned), software artifacts, data, and analyses. We present the workbench, describe its implementation, and report how it has been used by early adopters. Our initial case studies highlight 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 [5, 14]. A scientific community advances when its experiments are published, subjected to scrutiny, and repeated to determine the veracity of results. Repeated research not only helps to validate the conclusions of studies, but also to expand on previous conclusions and suggest new directions for 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. Data sets like those being collected in the networking community [2, 3, 6, 22] allow researchers to repeat analyses, but by themselves do not help researchers validate or repeat the data collection process. 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 should 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 being served by a variety of network testbeds: environments that provide resources for scalable and "real-world" experimentation. Some testbeds (such as Emulab [26]) focus on providing high degrees of control and repeatability, whereas others (such as PlanetLab [19]) focus on exposing networked systems to actual Internet conditions and users. Network testbeds differ from each other in many ways, but most have the same primary goal: to provide experimenters with access to resources. Once a person has obtained these resources, he or she usually 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 few features to help users repeat research in practice. Moreover, they provide little guidance toward making new experiments repeatable.

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 networked systems community. Emulab is our large and continually growing testbed: it provides 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 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 experiment management system called the *experimentation workbench*. The workbench is based on a new model of testbed-based experiments, one 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, capture experiment inputs and outputs, 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. The workbench combines these things, encompassing both an experiment management facility *and* an experiment execution facility. This is in contrast to typical scientific workflow management systems [27], which automate processes but do not manage the "laboratories" in which those processes operate. Thus, instead of saying that our 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 includes the resources necessary for re-execution.

The primary contributions of this paper are (1) the identification of *replayable research* as a critical part of future advances in networked systems; (2) the detailed presentation of our *experimentation workbench*, which is our evolving framework for replayable 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 [25].

This paper builds on our previous work [8] by detailing the actual workbench we have built, both at the conceptual level (Section 3) and the implementation level (Section 4). 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 5 and Section 6).

2 Background

Since April 2000, our research group has continuously developed and operated *Emulab* [26], a highly successful and general-purpose testbed facility and "operating system" for networked and distributed system experimentation. Emulab provides integrated, Web-based access to

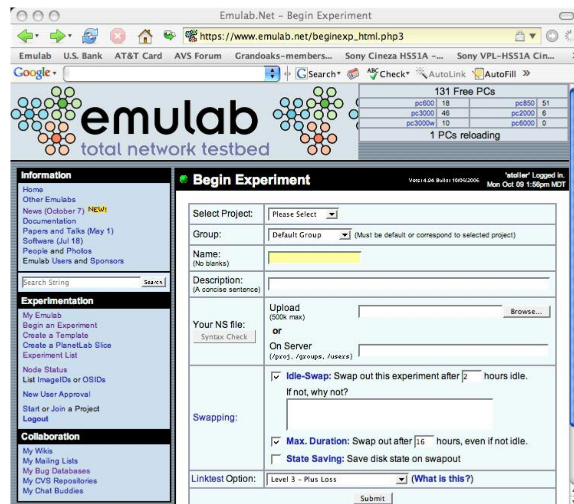

Figure 1: Emulab's Web interface

a wide range of 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 at Utah, 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 1. Once a user logs in, he or she can start an *experiment*, which is Emulab's central unit of operation. An experiment defines both a static and a dynamic configuration of a network as outlined in Figure 2. Experiments are usually described in an extended version of the *ns* language [11], but they may also be described through a GUI within Emulab's Web interface.

The static portion describes a network topology: a set of devices (nodes), the network in which they are contained, and the configurations of those devices and network links. The 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 for 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 a running process may signal the event at run time. Events can be assembled into event sequences as shown

```
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 start  # Do not wait for completion
    $do_client run    # Run client, wait till end
}]
$ns at 0.0 "$do_expt run"
$ns run
```

Figure 2: A sample experiment definition

in Figure 2. Each event in a sequence is issued when the preceding activity completes; some activities (like `start`) complete immediately whereas others (like `run` to completion) take time. 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 operate on nodes and links (e.g., to bring them up or down). Others are set up by a user as part of an experiment. These include the program agents mentioned above; *traffic generators*, which produce various types of network traffic; and *timelines*, which signal user-specified events on a timed schedule.

Through Emulab's Web interface, a user can submit an experiment definition and give 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 in Emulab provides persistent storage; this 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 six key ways that the original Emulab model of experiments breaks down for users.

1. An experiment entangles the notions of definition and instance. 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. In contrast to item 2 about relating experiments to one another, the issue here is relating user sessions to (possibly many) experiments and/or discrete tasks.

4. Data management is not handled as a first-class concern. Users must instrument their systems under study and orchestrate the collection of 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 also across experiments.

5. The old model does not help users 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, and so on that is utilized within the experiment. In contrast to item 3 about identifying multiple "units of work," here we are concerned about collecting the many components of an experiment definition.

6. The old model does not help users manage their studies over time. The artifacts and purpose of experimentation may 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.

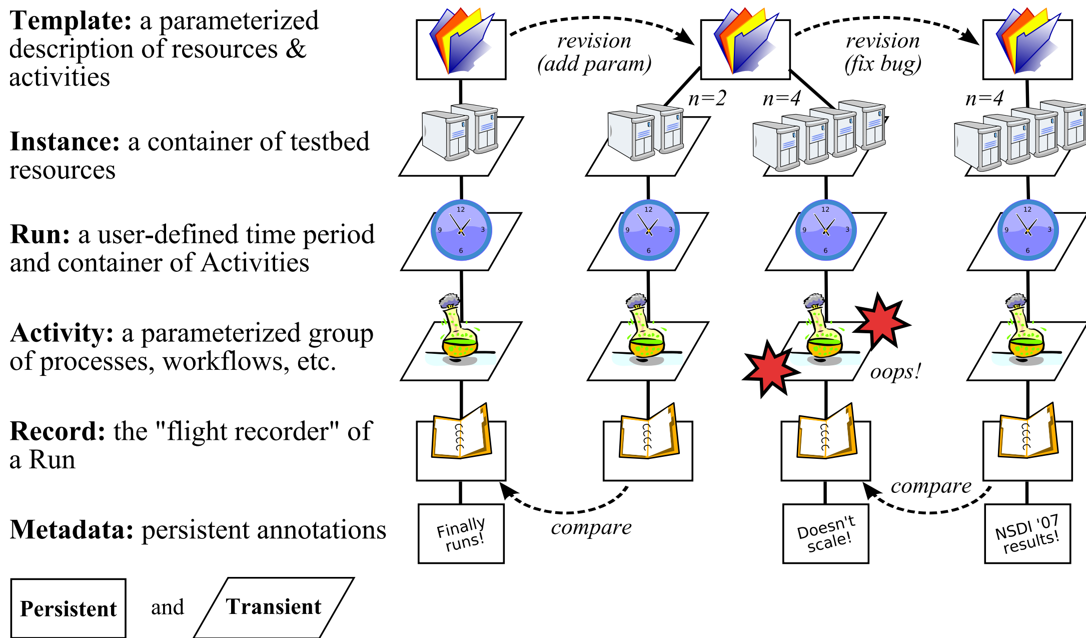

Figure 3: Summary of the workbench model of experimentation. The diagram illustrates an example of a user's experimentation over time. (The information is displayed differently in the workbench GUI, as shown in other figures.) The user first creates a template and uses it to get one set of results, represented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

3.2 Solution: refine the model

The problems above deal with the key concerns of replayable research. Addressing the shortcomings of Emulab's experiment model, 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 notion nevertheless fails to capture all the aspects of an experiment. Our new model divides the original notion of an experiment into parts, and then enhances those parts with new capabilities.

Figure 3 illustrates 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 components are persistent and are stored "forever" by the 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 at a high level. We have implemented the model (Section 4) and initial user experiences have been promising (Section 5), but they have also shown ways in which the model should be further refined (Section 6).

• A **template** is a repository for the many things that collectively define a testbed-based environment. It plays the "definition" role of Emulab's original experiment abstraction. Unlike an experiment, however, 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. A template may also contain other kinds data such as (reified) database tables and references to external persistent storage, e.g., datapositories [3] 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 a 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 **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, which is an extension 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, a user can specify parameter values when a template is instanti-

ated. These 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 transient entity: it plays the “resource owner” role of the original, monolithic experiment notion. The resources within an 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.

• A **run** is a user-defined context for activities. Conceptually, it 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 instance. For example, a user could separate the individual trials of a system under test; these trials could 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 as described below. Like a template instance, a run can have parameter values, which are specified when the 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. 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 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 account 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, 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 special observers such as packet recorders [9], filesystem monitors [8], and provenance-aware storage systems [15] can help determine the “extent” of a testbed-based activity. These can be automatically deployed by a testbed to drastically lower the experiment-specification burden for users.

Figure 4: Creating a new template

• Finally, **metadata** are 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. Through metadata, both users and the workbench itself can attach meaningful names, descriptions, and other mutable properties to templates and records.

4 Using the Workbench

In this section we present our current implementation of the workbench through an overview of its use. Our implementation extends Emulab to support replayable research, based on the conceptual model described in Section 3. Taken all at once, 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. The form is similar to the page for creating regular Emulab “experiments,” except that the controls related to swapping 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 [26].
- 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.

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 parameter name, *value* is the default value, and *desc* is an optional descriptive string. These will be presented 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.

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. (In the future, we will extend the workbench GUI so that users can add files to a template as part of the initial creation step.)

When a template is created, the workbench automatically creates a directory representing the template at a well-known place in Emulab's filesystem. One can think of this as a "checkout" of the template from the repository that is kept by the workbench. The part of the template that contains files is called the *datastore*, and to put files into a template, a user places those files in the template's `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 .
```

The user then "commits" 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, a commit results in a new template, and the workbench records that the new template is derived from the original. 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 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 filesystem or through the workbench's

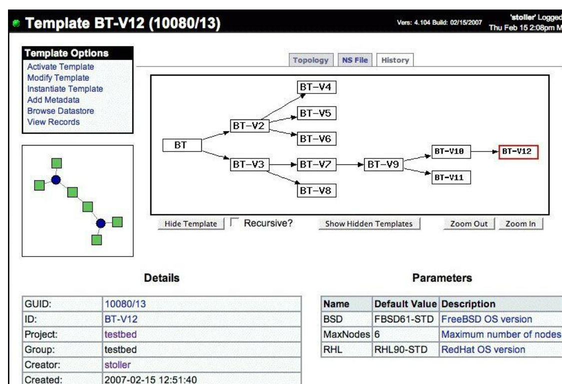

Figure 5: The workbench tracks and displays how templates are derived

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

Figure 6: Instantiating a template

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 study. The workbench tracks 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, a user can recall and inspect any template in the history. The workbench also provides controls that affect the tree display: e.g., individual nodes or subtrees can be elided.

4.2 Instantiating templates

Through the Web, a user can select a template and instantiate it. The form for instantiating a template is shown in Figure 6; it is similar to the form for swapping in a reg-

ular Emulab experiment, with two primary differences. First, the template form displays the template parameters and lets the user 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.

When a template is instantiated, a new 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 to files from the datastore. This feature also builds on existing Emulab practices. Users already know that Emulab creates a directory for each experiment they run; the workbench extends this by adding template-specific items, such as a copy 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* (Section 3.2) 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 to describing "pre-scripted" activities within our initial workbench.

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 illustrated in Figure 2, events for program agents typically refer to external scripts. The workbench makes it possible to encapsulate these files within templates, by putting them in a 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 to refer to files that are contained 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 executes the `template_record` command on some host within the template instance. For example, the following records an event to execute `client.sh`:

```
template_record client.sh
```

A second instance of `template_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.

4.4 Using records

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 a record.

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.
- *the logs* that were generated by testbed agents. These are written to well-known places, so the workbench can collect them automatically.
- *files* that were written to a special `archive` directory. Similar to the `datastore` directory described above, every template instance also has an `archive` directory in which users can place files that should be persisted.
- *the recorded dynamic events*, explained above.
- *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.

As described for templates, records are also stored in a Subversion repository that is internal to the workbench. We chose this design to save storage space, since we anticipated that the different records derived from a single template would be largely similar. Our experience, however, is that Subversion can be too slow for our needs when it is asked to process large data sets. We describe our experience with Subversion further in Section 6.

Inspecting records. The workbench stores records automatically and makes them available to users through

Figure 7: Viewing the records associated with a template

the Web. From the Web page that describes a template, a user can click on the *View Records* link to access the records that have been derived from that template. Figure 7 shows an example. Records are arranged in a two level hierarchy corresponding to template instances and runs within those instances. 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 provides a command-line program for exporting the full contents of a record.

Replay from records. The Web interface in Figure 7 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. 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 his or her activities. Whenever a new run begins, the user may specify new parameter values for that run. The set of parameters are those defined by the template; the user simply has an opportunity to change their values for the new run.

When a run stops, the workbench stops all the program agents and tracing agents within the run; collects the log files 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 begins, 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 scheduled events in the template's *ns* file will reoccur.

In addition to starting and stopping runs interactively, users can script sequences of runs using the event system and/or Emulab's XML-RPC interface. Through the event system, an experimenter can use either static (i.e., scheduled) or dynamic events to control runs. Using a command-line program provided by Emulab—or one written by the user in any language that supports XML-RPC—a user can start and stop runs programmatically, providing new parameter values via XML.

5 Case Studies

The experimentation workbench is a work in progress, and continual feedback from Emulab users is essential if we are to maximize the applicability and utility of the workbench overall. We therefore recruited several members of our own research group to use our prototype in late 2006. 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 project [7]. 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 of Flexlab 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 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, each pair containing one Emulab node and one "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, and directs Emulab to condition its network links to match the model. The details of these processes are complex and described elsewhere [7].

The original framework for a Flexlab experiment consisted of (1) an *ns* file, containing variables that control the topology and the specifics of a particular experiment, and (2) a set of scripts for launching the Flexlab services, monitoring application behavior, and collecting results. A 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 trial" script to start the Flexlab infrastruc-

ture. Thus, although all the files were managed via CVS, each experiment required by-hand modification of the *ns* file, and the Emulab experiment was not strongly 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 trial" script to tear down the Flexlab infrastructure and collect results. He or she would then analyze the results, and possibly repeat the start, stop, and analysis phases a number of times. The attributes of each trial 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 trial 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 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-trial scripts into the template datastore, thus making them part of the template. 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-trial 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 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. 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, the Flexlab developers can 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 for real tasks. For instance, while preparing their current paper [21], they used the workbench in a collaborative fashion to manage and review the data from dozens of experiments.

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—are included in the previously cited NSDI '07 paper about Flexlab [21].

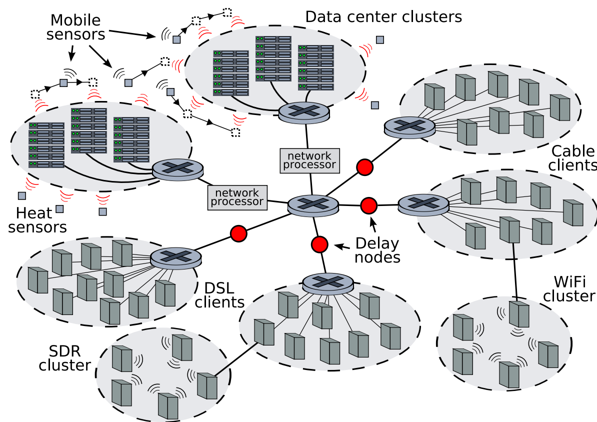

Figure 8: An example GHETE topology. GHETE utilizes the many types of emulated and actual devices in Utah's Emulab.

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 resource failures will be important for making the workbench robust and applicable to testbeds such as PlanetLab.

5.3 Study 3: application monitoring

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 the researcher in June 2006 to showcase Emulab's ability to handle large network topologies containing many types of nodes and links: wired and wireless PCs, virtual nodes, software-defined radio (SDR) nodes, sensor network motes, and mobile robots carrying motes. GHETE uses these node types inside an emulation of a distributed data center, such as the scenario shown in Figure 8. A data center consists of two (or more) 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 cluster 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. The clients in Figure 8 are connected to the data center through emulated DSL connections, emulated cable modem links, and true 802.11 wireless and SDR networks.

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 one-second intervals, and these measurements were stored in the database that was 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 [20] that analyzes 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 from his automated activities during the study. Each shows the effect of a given load-balancing strategy by plotting the absolute difference between the aggregate incoming bandwidths of the two service clusters over time. The spikes in the graphs correspond to times when one cluster was "shut down" due to simulated heat events. The results show that the second load balancer (Figure 10) yields more stable behavior, but the details of the load balancers are not our focus here. Rather, we are interested in how the workbench was used: did it help the GHETE developer perform his experiments and analyses more efficiently?

The GHETE developer said that the workbench was extremely useful in evaluating the load balancers 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 parameter values for each run within a template instance, software controlled by Emulab program agents

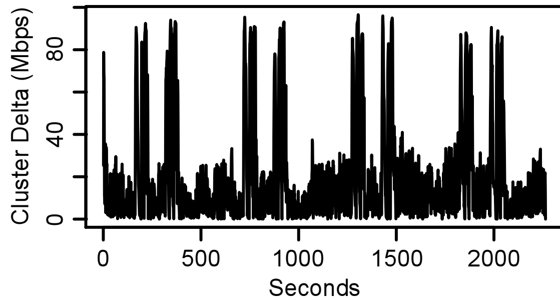

Figure 9: Measured difference in the incoming bandwidths of the two service clusters, using a simple load balancer

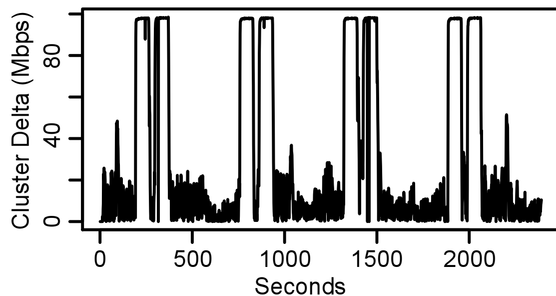

Figure 10: Measured difference in the incoming bandwidths of the two service clusters, using an improved load balancer

could be easily configured and reconfigured. This enabled rapid trials and reduced experimenter wait time. Second, via the per-template-instance database, the developer was able to analyze incoming data in real time within *interactive* R sessions. This is in addition to the post-processing analysis described above. He could also easily re-instantiate the database from any completed 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 Lessons Learned

In this section, we summarize two categories of "lessons learned" from the case studies and our other experiences to date with the prototype workbench. First, we discuss storage issues and the effect of our philosophy to have the workbench "store everything" by default. Second, we evaluate how well the workbench's model of experimentation actually describes the processes and relationships that are important to testbed-based research.

	Record size (KB)	**Stored in repo. (KB)**	**Elapsed time (s)**
BitTorrent (§5.2)	31,782	19,380	418
GHETE (§5.3)	72,330	20,240	861
Minimal template	88	48	48

Table 1: Space and time consumed for typical records in our case studies. The first column shows the size of the record, and the second shows the size of that record in the workbench's repository. The third shows the time required to stop a run, collect the data from all nodes, and commit the record. "Minimal template" describes the overhead imposed by the workbench.

6.1 Storage issues

It is essential for the workbench to store templates and records efficiently, because we intentionally adopted design principles that lead to high storage demands: "complete" encapsulation of experiments, proactive experiment monitoring, and saving history "forever" [8]. As described in Section 4, our prototype uses Subversion to store both templates and records in a space-efficient manner. We expected that the templates within a history tree would be quite similar to each other, and thus would be efficiently stored as deltas within Subversion. This argument also applies to the many records that arise from a template, but perhaps to a lesser degree. It was surprising to us, therefore, when we realized that the two primary lessons we learned about workbench storage were not about space, but about time and clutter.

First, as reported by our users, the prototype is too slow at processing large amounts of data. This is most commonly seen at the end of a run, when the workbench collects data from many nodes and commits the assembled record. The current workbench forces a user to wait until a record is complete before he or she can safely resume experimentation within a template instance. Table 1 describes typical records created from the templates used in our second and third case studies.

Our conclusion is that the workbench must collect and persist data more efficiently *from the user's point of view.* This deals partly with the amount of data collected, but more significantly, it deals with when that data is gathered. Both issues can be addressed by overlapping record-making with user activities in time. For example, the workbench may leave records "open" for a while. Log collection would occur in parallel with other user activities, i.e., new runs. Collected data would be held in escrow, and users would have a window in which to add or remove items from a record. When the window closes, user-specified rules would apply, and the record would be finalized.

To isolate record-gathering and user activities that run in parallel, we are investigating the use of branching filesystems. In particular, we have enhanced the

production-quality Linux "LVM" logical volume manager software [12]. Our extensions support the arbitrary branching of filesystem snapshots and greatly improve the performance of chained snapshots overall. A branching filesystem can prevent new runs from modifying the files created by previous runs. When deployed for the workbench repository (i.e., a shared filesystem), snapshots can naturally and efficiently encapsulate the contents of both templates and records. When deployed on the nodes within a template instance, it can remove the need for users to put all the files that they want saved into special directories for collection: all the files that are produced during a particular run can be identified as part of a particular snapshot. This "universal" archiving will raise issues of user intent, as sometimes users won't wish certain files to be persisted and/or restored. Such cases can be dealt with as described above, by leaving records "open" and applying user-specified rules.

The second lesson we learned was that, despite our vision of a workbench that stores data forever, users have a strong desire to delete data. Our users continually asked for the ability to destroy templates and records. These requests were not motivated by storage space, since we had plenty available. Instead, our users wanted to reduce cognitive clutter and maintain their privacy. They did not want to see their "junk" forever, and neither did they like the idea that their "mistakes" and other private activities might be forever viewable by others.

Although the records of old and failed experiments can be valuable for many reasons, it is clear that we need to provide a range of deletion options within the workbench. These will range from merely hiding templates and records to expunging them forever. Both present a challenge in terms of presenting undeleted objects in a consistent and complete way, and the latter puts a new requirement on the workbench data repository.

6.2 Model issues

The workbench is based on the model of experimentation described in Section 3. Based on our experience as users and administrators of Emulab, we designed the model to express notions and relationships that we knew needed to be captured over the course of testbed-based research. Overall, our experience with the new model has been very positive. Its abstractions map well to mechanisms that experimenters design for themselves, as we saw in our first case study (Section 5.1). Many of the problems that our users encountered were due to the immaturity of our prototype—i.e., unimplemented features—as opposed to problems with the model.

On the other hand, the users in our first and second cases studies also had problems that *were* traceable to shortcomings of the model. In the first study, there was confusion about the relationships between runs: does a run start with the environment (e.g., filesystem and database) that was left by the preceding run, or does each run start afresh? In the second study, the experimenter wanted to substantially change his network topology between runs, and he had trouble with the workbench when nodes failed unexpectedly. All these issues stem from the *life cycle concerns* of objects and abstractions that are not clearly represented in our new model of experimentation. We can and will refine the model to address the issues that emerged during our case studies.

We always expected to refine our model as we gained experience with the workbench. What we learned is that many of the problems of our current model stem from a single key difference between the requirements of our workbench and the requirements of experiment management systems for other scientific domains. It is this: Unlike systems that track workflows and data within static laboratories, the workbench must manage a user's activities *and* the user's dynamic laboratory at the same time.

The user's laboratory contains his or her testbed resources, which have dynamic state and behavior. The life cycles of the user's laboratory and experimental activities are separate but intertwined. Our workbench models the separation—it distinguishes template instances from runs—but not the interactions that should occur between these two levels of testbed use. For example, it is not enough for template instances to start and stop; they should also be able to handle resource failures on their own, via user-specified procedures. Instances also need to communicate with runs to coordinate the handling of failures with users' experimental activities. Some existing systems (e.g., Plush [1]) model concerns such as these, and their models will help to guide the evolution of our workbench.

7 Related Work

The workbench is an experiment management system, and there is a growing awareness of the need for such systems within the networking research community. Plush [1], for instance, is a framework for managing experiments in PlanetLab. A Plush 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 testbed nodes. Plush thus provides features that are similar to those provided by Emulab's core management software, which utilizes extended *ns* files and provides a Web-based GUI. Our workbench builds atop these services to address higher-level concerns of experimentation man-

agement: 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 [24] is a second experiment management system 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.

Many scientific workflow management systems have been developed for computational science, including Kepler [13], Taverna [17], Triana [23], VisTrails [4], and others [27]. Many of these are designed for executing distributed tasks in the Grid. Our workbench has much in common with these systems in that the 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 networked systems 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. 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 networked systems.

Many experiment management systems have also been developed for domains outside of computer science. For instance, ZOO [10] 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 [16], a popular commercial product that interfaces with many scientific instruments. It has been used as the basis of several "virtual laboratories," and like Emulab, LabVIEW can help users manage both real and simulated devices. 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.

8 Conclusion

Vern Paxson described the problems faced by someone who needs to reproduce his or her own network experiment after a prolonged break [18]: "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*, which pairs repeatable experiments with the testbed facilities that are needed to repeat and modify experiments in practice. We have presented the design and implementation of our experimentation workbench that supports replayable research for networked systems, and we have described how early adopters are applying the evolving workbench to actual research projects. 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 incorpo-

rates and helps to automate the community practices that Paxson suggests [18]: 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, David Johnson, and Robert Ricci 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. We thank David Andersen for his work on the datapository and Prashanth Radhakrishnan for his work on branching LVM. Finally, we thank the anonymous NSDI reviewers and our shepherd, Rebecca Isaacs, for their many constructive comments.

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

References

[1] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. PlanetLab application management using Plush. *ACM SIGOPS OSR*, 40(1):33–40, Jan. 2006.

[2] M. Allman, E. Blanton, and W. Eddy. A scalable system for sharing Internet measurements. In *Proc. Passive and Active Measurement Workshop (PAM)*, Mar. 2002.

[3] D. G. Andersen and N. Feamster. Challenges and opportunities in Internet data mining. Technical Report CMU–PDL–06–102, CMU Parallel Data Laboratory, Jan. 2006. `http://www.datapository.net/`.

[4] L. Bavoli et al. VisTrails: Enabling interactive multiple-view visualizations. In *Proc. IEEE Visualization 2005*, pages 135–142, Oct. 2005.

[5] B. Clark et al. Xen and the art of repeated research. In *Proc. FREENIX Track: 2004 USENIX Annual Tech. Conf.*, pages 135–144, June–July 2004.

[6] Cooperative Association for Internet Data Analysis (CAIDA). DatCat. `http://www.datcat.org/`.

[7] J. Duerig et al. Flexlab: A realistic, controlled, and friendly environment for evaluating networked systems. In *Proc. HotNets V*, pages 103–108, Nov. 2006.

[8] E. Eide, L. Stoller, T. Stack, J. Freire, and J. Lepreau. Integrated scientific workflow management for the Emulab network testbed. In *Proc. USENIX*, pages 363–368, May–June 2006.

[9] Flux Research Group. Emulab tutorial: Link tracing and monitoring. `http://www.emulab.net/tutorial/docwrapper.php3?docname=advanced.html#Tracing`.

[10] Y. E. Ioannidis, M. Livny, S. Gupta, and N. Ponnekanti. ZOO: A desktop experiment management environment. In *Proc. VLDB*, pages 274–285, Sept. 1996.

[11] ISI, University of Southern California. The network simulator - ns-2. `http://www.isi.edu/nsnam/ns/`.

[12] A. J. Lewis. LVM HOWTO. `http://www.tldp.org/HOWTO/LVM-HOWTO`.

[13] B. Ludäscher et al. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, Aug. 2006.

[14] J. N. Matthews. The case for repeated research in operating systems. *ACM SIGOPS OSR*, 38(2):5–7, Apr. 2004.

[15] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proc. USENIX*, pages 43–56, June 2006.

[16] National Instruments. LabVIEW home page. `http://www.ni.com/labview/`.

[17] T. Oinn et al. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, Aug. 2006.

[18] V. Paxson. Strategies for sound Internet measurement. In *Proc. 4th ACM SIGCOMM Conference on Internet Measurement (IMC)*, pages 263–271, Oct. 2004.

[19] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. *ACM SIGCOMM CCR (Proc. HotNets-I)*, 33(1):59–64, Jan. 2003.

[20] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2006. `http://www.r-project.org/`.

[21] R. Ricci et al. The Flexlab approach to realistic evaluation of networked systems. In *Proc. NSDI*, Apr. 2007.

[22] C. Shannon et al. The Internet Measurement Data Catalog. *ACM SIGCOMM CCR*, 35(5):97–100, Oct. 2005.

[23] I. Taylor et al. Distributed computing with Triana on the Grid. *Concurrency and Computation: Practice and Experience*, 17(9):1197–1214, Aug. 2005.

[24] Y. Wang, M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Automating experimentation on distributed testbeds. In *Proc. Conf. on Automated Software Engineering (ASE)*, pages 164–173, Nov. 2005.

[25] K. Webb et al. Implementing the Emulab-PlanetLab portal: Experience and lessons learned. In *Proc. Workshop on Real, Large Dist. Systems (WORLDS)*, Dec. 2004.

[26] B. White et al. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, Dec. 2002.

[27] 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.