Automating Experimentation on Distributed Testbeds

Yanyan Wang[†] Matthew J. Rutherford[†] Antonio Carzaniga[†,‡] Alexander L. Wolf[†,‡]

[†]Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430

[‡]Faculty of Informatics
University of Lugano
6900 Lugano, Switzerland

ABSTRACT

Engineering distributed systems is a challenging activity. This is partly due to the intrinsic complexity of distributed systems, and partly due to the practical obstacles that developers face when evaluating and tuning their design and implementation decisions. This paper addresses the latter aspect, providing techniques for software engineers to automate the experimentation activity. Our approach is founded on a suite of models that characterize the distributed system under experimentation, the testbeds upon which the experiments are to be carried out, and the client behaviors that drive the experiments. The models are used by generative techniques to automate construction of the workloads, as well as construction of the scripts for deploying and executing the experiments on distributed testbeds. The framework is not targeted at a specific system or application model, but rather is a generic, programmable tool. We have validated our approach by performing experiments on a variety of distributed systems. For two of these systems, the experiments were deployed and executed on the PlanetLab wide-area testbed. Our experience shows that this framework can be readily applied to different kinds of distributed system architectures, and that using it for meaningful experimentation, especially in large-scale network environments, is advantageous.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Experimentation

Research supported in part by the National Science Foundation, Defense Advanced Research Projects Agency, and Army Research Office under agreement numbers ANI-0240412, F30602-01-1-0503, and DAAD19-01-1-0484.

ASE'05, November 7–11, 2005, Long Beach, California, USA.

Keywords

Distributed systems, wide-area testbeds, PlanetLab, experiment automation

1. INTRODUCTION

This paper addresses the problem of experimenting with highly distributed systems. We use the term *highly distributed system* to refer to a system capable of delivering a service to many clients through a large number of distributed access points. This is in contrast to a traditional client/server system, where a single access point offers service to multiple distributed clients. A highly distributed system usually consists of a network of components, executing independent and possibly heterogeneous tasks, that collectively realize a coherent service. Examples of such systems are various forms of communication systems, application-level overlays, networks of caching servers, peer-to-peer file-sharing systems, distributed databases, replicated file systems, and distributed middleware systems in general.

Engineering a highly distributed system is a challenging activity. The difficulties are due in part to the intrinsic complexity of large, distributed architectures and protocols, and in part to the practical obstacles that one faces in evaluating and tuning alternative designs and implementations. Difficulties of the first kind arise early on in the development process, where analytical methods and simulations can offer valuable guidance to the engineer. By contrast, the latter kind of difficulties are typical of the later stages of development, where only systematic, repeated experimentation with executable prototypes in realistic execution environments can yield accurate results. Unfortunately, the scale, heterogeneity, and dynamism of highly distributed systems make it difficult to efficiently conduct these experiments manually, and existing tools for experimenting with distributed systems are primarily oriented toward much simpler scenarios, such as client/server.

The work described here focuses on this latter aspect. Specifically, we have developed a comprehensive framework to help software engineers manage and automate experiments with highly distributed systems performed on distributed testbeds. The framework covers a simple two-phase process of *workload generation* and *experiment deployment and execution*.

A distributed system workload is a partially ordered list of service calls from clients to system access points. Typically, such workloads are created either by logging the actual behavior of an application during a particular run or by positing a probabilistic distribution of service calls. The former

requires an application to pre-exist the experiments, while the latter requires a careful analytical analysis of application behavior. We introduce an additional, complementary approach in which the engineer first performs an offline simulation of expected client behavior and records the resulting service calls. Then, during an actual run of an experiment, the service calls are issued as live stimuli to the subject system; the same set of recorded service calls can be reused in different experimental scenarios. Simulation requires neither the availability of an application nor the results of a probabilistic analysis, although both can be used in the construction of the simulation.

Experiment deployment and execution applies a workload to a system and gathers run-time data. To achieve this, the subject system and the workload are first deployed across an experimentation testbed. Next, the system is started, and then stimulated by the execution of service calls at the times and locations dictated by the workload. When the experiment terminates, logged output and diagnostic data are collected from across the testbed to facilitate data analysis and reduction. Deployment and execution are driven by scripts. But rather than asking engineers to provide these scripts, we automate their construction from a suite of more easily configured and higher-level models of the system, the testbed, and the desired experiment.

Our implementation of the framework is called Weevil.[1] Weevil is not targeted at a specific system or application model, but is a generic, programmable tool. We have used Weevil to experiment with a variety of highly distributed systems, both on a local-area distributed testbed and on a wide-area distributed testbed, PlanetLab [14].[2] The PlanetLab testbed, by intention, exhibits many of the challenges of a true deployment: faulty nodes, faulty communication, and unpredictable delays.

Our early experience, reported here, demonstrates that Weevil contributes a useful and powerful new tool to the engineers of highly distributed systems. In particular, we have used Weevil to:

- model two different publish/subscribe communication systems (Siena [3] and Elvin [15]), a mobility service for publish/subscribe clients (MobiKit [2]), two different peer-to-peer file sharing systems (Freenet [6] and Chord [16]), and a composite web-cache system (Squid proxies [8] and Apache web servers [11]);
- model two different deployment and execution environments (a local-area testbed and the PlanetLab wide-area testbed);
- reproduce and broaden the results of a published study on cooperative web caching [20] without incurring the cost and difficulty of collecting additional live trace data;
- validate experimentally the results of an analytical analysis of Freenet [5] that predicted the behavior of a new routing algorithm; and
- provide a more realistic experimental analysis of scalability in Chord than that of a published study [16] by performing an experiment that used a similar number of distributed components deployed over an order of magnitude more machines.

There are two important lessons to take away from these experiences. First, we were able to use Weevil to gain useful and novel results and, second, Weevil made the process of gaining those results substantially less costly by allowing us to leverage automation. In principle, the same results could be achieved through manual means, but in practice the time and effort available to an engineer (or researcher, for that matter) are limited. Using Weevil we were able to carry out multiple experiments on large numbers of elements while managing the exploration of broad experimental configuration spaces. We found Weevil's ability to rapidly propagate parameter and configuration changes, as well as rerun experiments in the face of node failures, especially important.

In the next section we provide some terminology and introduce a simple example. Our simulation-based approach to workload generation is presented in Section 3. Section 4 describes our use of models and generative techniques to support experiment deployment and execution. Our experiences with Weevil are discussed in Section 5. Some related technologies are reviewed in Section 6 and we conclude in Section 7 with a discussion of future work.

2. BACKGROUND

A particular experiment is related to three primary concepts: the system under experimentation (SUE), the testbed, and the actor. We use the term "actor" here to avoid confusion with the overloaded term "client". An actor maps a client to its system access point and stimulates the SUE as dictated by the client's portion of the workload. An experiment consists of (1) selecting or generating a workload, (2) configuring and deploying the actors and the SUE, (3) running the actors and the SUE, and (4) returning data for review. Weevil supports these four activities through a central-controller architecture in which a master manages the process. The master generates control scripts, which are deployed together with the actors and the SUE on the testbed.

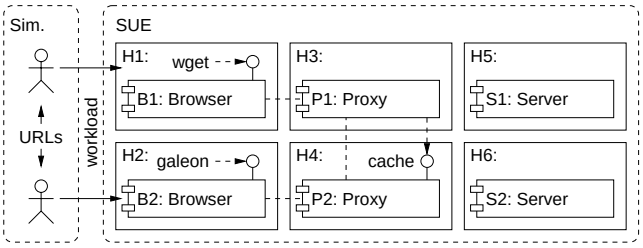

Figure 1: Deployment Diagram of Web-Proxy Experiment

In the next two sections we refer to an example experiment aimed at studying the performance of web proxies. Figure 1 shows a deployment diagram of the SUE and testbed used in this experiment. There are six components mapped onto six hosts. The components are of three kinds: browsers, web proxies, and web servers. It is important to note that although we are primarily interested in evaluating the perfor-

[1] http://www.cs.colorado.edu/~ywang/weevil/

[2] PlanetLab is an overlay network currently consisting of over 500 nodes located at over 250 sites around the world. Its stated purpose is to provide an "open platform for developing, deploying, and accessing planetary-scale services". See http://www.planet-lab.org/.

mance of the proxies, the browsers and servers are included as part of the SUE, since they represent key elements of the operating environment. Figure 1 also shows two actors communicating with the SUE. The two actors exhibit an interdependent behavior (during workload generation, not experiment execution), engaging in out-of-band communication (with respect to the SUE) by exchanging URLs.

3. WORKLOAD GENERATION

A common practice in software experimentation is to generate a workload on the basis of a statistical model that abstracts usage patterns of the SUE. This approach is concise and efficient. However, usage patterns often vary widely based on context, and a statistical model offers only limited expressiveness. Also, sufficient data must be available to create an accurate statistical model of an existing behavior.

As a more general, complementary approach, Weevil supports an operational technique that models usage behavior directly. Specifications for an operational model could come from, for example, empirical traces [17], detailed user behavior profiles [12], or an experimentation plan in which specific usage scenarios are described. Our idea is to give software engineers the ability to quickly and easily express their specific system usage scenarios or user behaviors to create a diversity of workloads.

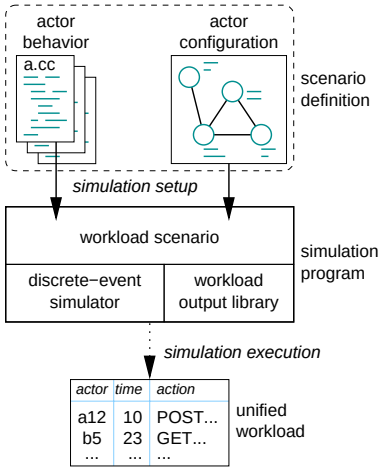

Figure 2: Simulation-Based Workload Generation

The workload generation process is illustrated in Figure 2. It allows the engineer to model one or more types of actor behaviors as *programs* written in a common programming language, such as C++, supported by a workload-generation library. The actors may therefore execute arbitrary functions and maintain arbitrary state. After programming the actor behavior types, the engineer populates a scenario consisting of actor instances specified in the actor configuration. The actor behavior types and the actor configuration make up a workload scenario definition. A workload scenario is then translated into an executable simulation program that is linked with the workload-generation library and executed to produce the desired workload.

The workload consists of all interactions between actors and the SUE, which are recorded by a special output function provided by the Weevil workload-generation library. These interactions represent service calls that are applied to the SUE during the actual experiment execution. Thus, we are using a discrete-event simulator to simulate actor behaviors, and capture the service calls made to the SUE as a reusable and reconfigurable workload that can be applied in multiple experimental scenarios.

In practice, actor behavior is encapsulated in a subclass of the `WeevilProcess` class, which is itself an extension of the `Process` class provided by the SSim discrete-event simulation library.[3] The library supports message communication between instances of `Process`, so behavior programs may specify interactions with other actors as well as interactions with the SUE.

In the web-proxy example, we generate a workload by simulating two humans browsing two sites and randomly picking URLs that are known to exist. Additionally, each human periodically recommends a URL to the other, who immediately requests the recommended URL upon receiving the recommendation. Modeling such a behavior as a program is straightforward for a software engineer.

We have used simulation-based workload generation to model a wide variety of actor behaviors. Examples are detailed elsewhere [18].

Our motivation for developing the simulation-based workload generator is its inherent flexibility and scalability. It is flexible in two dimensions. First, it can be immediately used to program workload generators based on statistical models. In fact, those generators reduce to scenarios with independent stochastic processes. Second, because it is fully programmable, it offers a natural way to represent complicated actor behaviors at any level of abstraction. It allows for an easy and compact specification of interdependent dynamic client behaviors that may result in complex and interesting workloads for collaborative activities.

Simulation-based workload generation is scalable in the sense that it can seamlessly deal with very complex scenarios, consisting of a multitude of interacting actors, executing over long periods of (virtual) time. In fact, this is precisely what simulation engines are designed to do. This ability to scale up is particularly beneficial because it allows an engineer to produce workloads in which complex collective behaviors emerge from simple individual behaviors.

4. DEPLOYMENT AND EXECUTION

Weevil is a generic, programmable tool for performing experiment deployment and execution. The overall process is depicted in Figure 3. Actions are represented by rectangles and are labeled by circled numbers. Input and output data for those actions are represented by ovals. Dark ovals represent input models provided by the engineer. White ovals represent control scripts and data files generated by Weevil. The cross-hatched ovals represent data generated by the SUE during an experiment. Solid arrows represent normal input/output data flow, whereas dotted arrows represent the execution of scripts.

We have taken an automated, model-based approach to the design of Weevil, whereby generative techniques are used to transform experiment configuration directives into an experiment management framework. This provides three main advantages over manual approaches: (1) engineers are relieved of the burden of creating and maintaining a large volume of experiment control scripts, and instead must only deal directly with a relatively concise set of configuration

[3]http://www.cs.colorado.edu/serl/ssim/

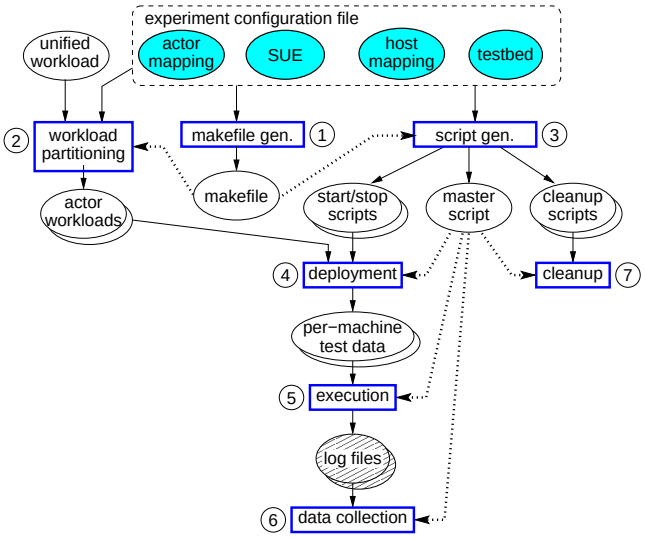

Figure 3: Weevil Experimentation Process

parameters; (2) models can be shared among experiments and easily tweaked when experiments must be changed; and (3) Weevil's generative capabilities transparently handle much of the complexity brought about by a system's or a testbed's scale and heterogeneity.

4.1 Configuration Modeling

For each experiment, Weevil must be provided with a workload together with experiment configurations for two primary models: the SUE and the testbed. Additionally, mappings between the SUE and the testbed, and between actors represented in the workload and the SUE, must be provided. These configuration models are represented by the dark ovals along the top of Figure 3. In the current version of Weevil, they are programmed in GNU m4 [1] by calling Weevil-defined declaration macros to instantiate the model elements. In other words, the declaration macros will define a set of property macros serving as properties of an experiment. Other than the reserved properties described here, Weevil allows system-specific properties to be assigned. All these property macros can be used as parameters in other declaration macros. They are resolved during script generation using m4's macro expansion. Weevil supports the engineer during this activity by performing extensive checks on the syntax and consistency of the configurations, and by providing detailed error messages about any problems encountered.

SUE Model. The conceptual model of an SUE is shown in Figure 4. As this figure shows, an SUE is comprised of typed *Component*s, *Relation*s between them, and an *Order* in which to start up the components.

In general, an SUE can consist of different types of components, such as the browsers, proxies, and servers in the example of Figure 1. Each component is declared as an instance of a *ComponentType*. *ComponentType* has *startScript* and *stopScript* attributes, and optionally a *config* attribute that contains the contents of a configuration file. This design allows common attributes to be shared among all components of a type. For different experiments targeting the same system, an engineer would typically need to make minor changes to other entities without having to modify *ComponentType* attributes. The attributes *processing* and *output* are used to specify post-processing of experiment output: a *processing* script is first executed on the component log files, and then the *output* of the script is copied back from each component's workspace to the master.

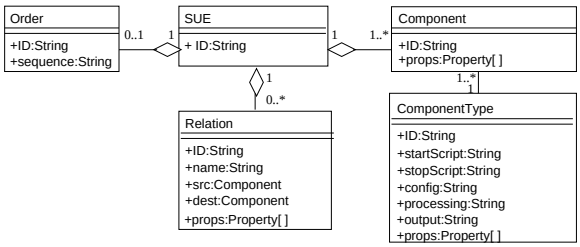

Figure 4: SUE Conceptual Model

The *Relation*s contained in an SUE model are used to represent any binary associations between components. These are optional and entirely system specific. For instance, in Figure 1 there are three relations (shown as dashed lines): component P1 is a "proxy" for B1, P2 is a "proxy" for B2, and P1 is a "peer" of P2. In general, relations are used in situations where one component references properties of another component for its execution.

Order entities are used to represent the necessary or preferred order in which to start the components. This is optional and entirely system specific, since some SUEs require certain components to be ready before others. For example, Siena requires a parent server to be ready before its children can start, but the children can start in any order. Similarly, a bootstrap Chord instance needs to be available before any other nodes can join a Chord network.

Testbed Model. Weevil makes minimal assumptions about the testbed. It only requires an account on each testbed host accessible through user-level remote shell access. As shown in Figure 5, a *Testbed* has an identifier and a collection of *Host*s.

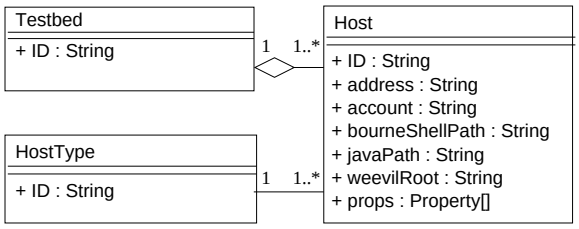

Figure 5: Testbed Conceptual Model

Each host in a testbed is an *account* on a network *address*. In PlanetLab the account is actually the PlanetLab slice name to which the engineer is assigned. Weevil uses the Bourne shell for experiment framework execution. Thus, each *Host* has the attribute *bourneShellPath* to provide its local path to the program *sh*. The *javaPath* attribute is needed if an actor is implemented in Java, as described below. *weevilRoot* specifies the workspace on the host assigned for the experiment.

To support deployment on heterogeneous testbeds, Weevil includes a *HostType* entity that is used to partition the hosts into categories needed for each software binary package.

Mappings. The two models described above are designed to be largely independent of each other and of the workload to be used during an experiment. This gives the engineer a fair amount of flexibility in composing experiments. The connection among the three is specified using two mappings.

The first mapping associates the SUE and the testbed by simply specifying on which host of the testbed each component of the SUE should reside.

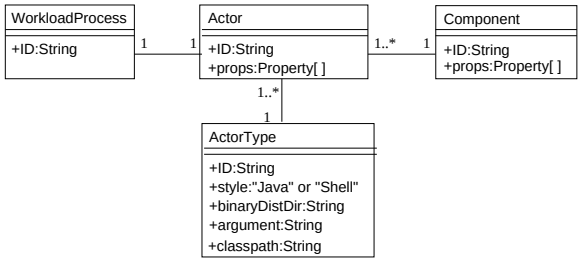

Figure 6: Actor-Mapping Conceptual Model

The second mapping, shown in Figure 6, associates the workload and the SUE. Each workload process is mapped to a single component through an *Actor* that is declared as an instance of an *ActorType*. *ActorType* represents the implementation of the actors using the same system APIs. An actor is a system-specific program that understands how to stimulate the SUE as dictated by the workload. Weevil provides a library to support its implementation in Java or as a shell script.

4.2 Setup and Script Generation

Given a set of configuration models, the engineer can now "compile" the experiment configurations into the framework scripts that will be used for experiment deployment and execution. First, the configurations are checked for consistency, and a per-experiment Make file is generated to control the rest of the process (shown as action 1 in Figure 3). Next, the overall workload is tailored according to the experiment configuration (action 2). For instance, in the web-proxy example, the network addresses of the web-server components are not known when the workload is generated, but are rather dictated by the component-to-host mapping. Also performed as part of this action is the partitioning of the overall workload into per-actor workloads. Finally, a start script, a stop script, and a cleanup script are generated for each component in the SUE, and a master control script is generated to manage the execution of the experiment (action 3).

4.3 Deployment and Execution

At this point, the engineer can perform an experiment by simply executing the master control script. The master control script deploys the components of the SUE, per-actor workloads, actors, and control scripts to the hosts (action 4), then starts all the components and actors (action 5). By estimating the round-trip time between the master host and testbed hosts, the master script intelligently decides when to have actors begin processing their workloads. The master script waits for all actors to complete processing their workloads and then causes execution of the stop scripts for each of the components. After all the components terminate, post-processing scripts are executed and output is copied back to the master machine (action 6). Finally, the testbed machines are cleaned up as necessary (action 7).

We dedicated significant effort to designing a robust master control script that allows an experiment to be run as efficiently and reliably as possible. For example, Weevil copies files from the master to the testbed hosts in parallel, thereby reducing the overall deployment time significantly. Additionally, the master script takes great care to handle the many error conditions that may arise during experiment deployment and execution in complicated distributed environments, such as PlanetLab. In particular, Weevil's master control script detects and reports errors as early as possible to prevent distorted experimental results and wasted resources. Moreover, the script is broken into discrete work units that are isolated transactionally so that the setup and deployment steps of an experiment need not always be repeated when errors occur.

5. EXPERIENCE

Our experience with Weevil has been focused on evaluating and improving Weevil's applicability to various systems, workloads, and testbeds, as well as Weevil's utility in automating the experimentation process. Specifically, we are asking the following research questions.

- *Versatility of the system models:* Is Weevil applicable to a wide variety of highly distributed systems?
- *Fidelity of the workload models:* Is Weevil's workload-generation method faithful to specific problems and scenarios?
- *Scalability of the deployment and execution mechanisms:* Is Weevil capable of handling large-scale experiments and testbeds?
- *Utility of the automation features:* Does Weevil provide an effective cost savings in automating the experimentation process?

Below we describe our experience with respect to each individual question.

5.1 Model Versatility

To evaluate Weevil's breadth of applicability, we modeled and ran experiments on six, quite different highly distributed systems: Siena [3], a publish/subscribe service implemented through a network of servers; Elvin [15], a publish/subscribe middleware system that supports the federation of servers; MobiKit [2], a mobility framework for distributed publish/subscribe systems that is implemented through a proxy-client mechanism; Freenet [6], a peer-to-peer file sharing system; Chord [16], a distributed value/location lookup service based on distributed hashing; and a composite web-cache system made up of Squid proxies [8] and Apache web servers [11].

The six systems provide a representative sampling from the broad spectrum of highly distributed systems. Each has unique characteristics that require special consideration when configuring experiments. For example, Weevil's script generation was designed to support systems like Siena that are controlled by command-line parameters, as well as systems like Freenet that are controlled by configuration files. In the end, Weevil was able to accommodate many such system-specific features.

An extensive discussion of how we used the features of Weevil to model and exercise the six systems is beyond the scope of this paper, but is available elsewhere [18]. Nevertheless, we provide some additional information for each of Squid/Apache, Chord, and Freenet as part of our discussion of the other three research questions.

5.2 Workload Fidelity

To verify that Weevil's simulation-based workload generation method can faithfully create realistic workloads, we developed workloads representing the web-cache client behavior described in a study on cooperative web caching [20]. According to the study, cooperative caching improves cache hit rate rapidly with smaller populations, while it is unlikely to provide significant benefit for larger populations. The study was based on traces collected from multiple live proxies at the University of Washington and Microsoft Corporation, data that are usually difficult to obtain.

Based on the analysis provided by the trace study, we created a workload by modeling and simulating the behavior of several actor groups, each one representing an independent organization in the study. The actors within one group use the same proxy and tend to have similar interests, resulting in a higher level of intra-group communication than inter-group communication. Thus, the population of actors in a scenario with fewer groups is overall more homogeneous in its web-access behavior. We generated several different workloads by executing the simulation program with different actor population configurations.

The workloads were then applied to a distributed web-caching system made up of several Apache servers and three or six Squid proxies, depending on the number of actor groups modeled in the workload. The actors were implemented as a shell script that parses each work line in the workload and hands off the execution of the actions to a program, *squidclient*, that is provided as part of Squid's distribution. We conducted the experiments on a local-area testbed consisting of six hosts running the Linux or FreeBSD operating system. Weevil supports the deployment of appropriate component software binary distribution onto the heterogeneous testbed through the *hostType* attribute of each host. Using a local-area testbed is reasonable in this case, since the cache hit rate is unrelated to network latency.

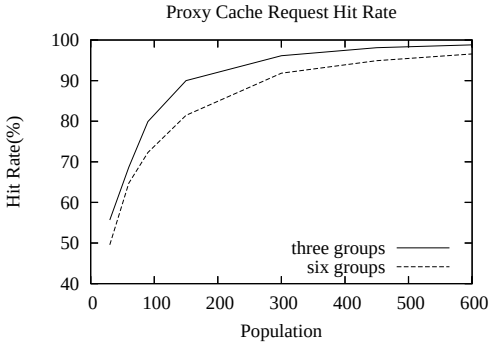

Figure 7: Cache Hit Rate vs. Client Population

Figure 7 shows the hit rates for experiments with three actor groups (proxies) and experiments with six actor groups (proxies). Each of the plotted lines has an inflection point with a steep increase in request hit rate below and a shallower increase above. The experiments with three groups have higher hit rates, which results from their more homogeneous behavior.

These results reproduce and extend the analysis provided in the published study. They demonstrate our ability to model a complex, empirically derived behavior. Moreover, we were able to set up these experiments within several hours, with the first requiring the most effort, since it included the time to model the system and testbed. Once these elements were in place, creating and executing additional experiments involved only simple adjustments to parameters.

5.3 Framework Scalability

Since highly distributed systems are targeted at large-scale network environments, we cannot evaluate them fully without experimenting on large-scale testbeds. This is the ultimate purpose of Weevil. To determine how Weevil performs in the large, we used it to conduct experiments with Freenet and Chord on PlanetLab.

Freenet Experiments. Freenet is a peer-to-peer file-sharing system. It has been undergoing a redesign of its core architecture: the "Next Generation (NG) Routing Algorithm" is intended to replace its "Classic Algorithm". According to Clarke's analysis of the two algorithms [5], the NG routing algorithm makes Freenet nodes much smarter about deciding where to route information when a new request is received. Based on that analysis, the NG routing algorithm should exhibit better scalability and show improved performance for most requests.

To verify this, we conducted a series of experiments in which we varied the number of Freenet nodes and their geographical separation. In each experiment, a number of Freenet instances are started across the testbed. Each of them is located on an individual PlanetLab host. An actor's behavior is to inject 20 files into the Freenet overlay and then to issue 16 requests for retrieval of randomly chosen file names from all the injected files. To avoid any unwanted effects due to file size, the length of all files in the experiments is fixed at 200 bytes. The requests are issued one-by-one with a random interval.

The workloads for different experiments are to be produced by different numbers of actors using the same behavior model just described. We first programmed the actor behavior model. Then for each different experiment, we simply adjusted the number of actor declarations in the actor configuration. Weevil created the simulation program for all the actors, and generated the unified workload and the files to be injected.

As all the components in the SUE are Freenet nodes, we declared a single component type called *FreenetNode*. A feature of Freenet, which is quite common among other distributed systems, is that its components are controlled by configuration files rather than command-line parameters. This requires Weevil to customize a configuration file on a per-component basis. As shown in Figure 4, *ComponentType* contains a *config* attribute that is the content of its configuration file. This content is retrieved during setup using m4's include mechanism. For Freenet, we add macros into the default configuration file that are expanded during script generation. The following snippet from our template Freenet configuration file shows how the `listenPort` parameter is

set by Weevil:

```
listenPort = 'WVL_Component_'WVL_Component_ID'_ListenPort'
```

Each Freenet node has a `ListenPort` property assigned to it that is stored in a property macro of the form: `WVL_Component_<ID>_ListenPort`. During setup, the macro `WVL_Component_ID` is defined in turn to the identifier of each component currently being processed. So, the macros on the right-hand side of the assignment expand to be the listen port for the component being processed. Other parameters in the template configuration file are set similarly. As a result, the template configuration file is customized for each component in the SUE.

We did not include *Order* and *Relation* configurations in the SUE model since the order to start up Freenet nodes does not matter and the relations between Freenet nodes are not explicitly configured. Therefore, the only configuration differences between experiments are the number of component declarations and their property definitions.

We applied the same workloads and experiment configurations to versions of Freenet implementing the two routing algorithms. Our experiment sizes ranged from 10 to 120 Freenet nodes, each deployed to an individual PlanetLab host. For each experiment, we measured the times required to locate and download the requested files.

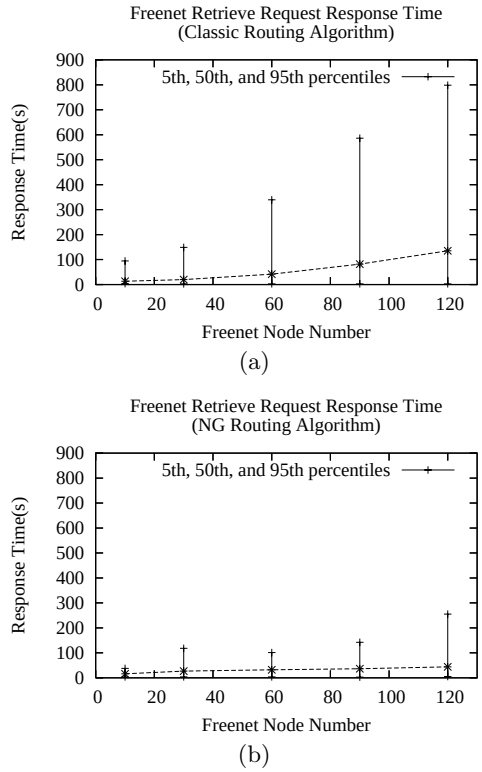

Figure 8: Freenet Retrieve Latency

Figures 8a and 8b show the median, the 5th, and the 95th percentile of retrieve latency for the Classic and NG routing algorithms, respectively. The median latency increases from 13s to 135s for the Classic algorithm, and from 16s to 44s for the NG algorithm. Obviously, the Classic algorithm has a significantly higher retrieve latency rate on average. This validates our expectations of the relative scalability of the two routing algorithms. Additionally, Figure 8 shows that the Classic algorithm exhibits a larger standard deviation than the NG algorithm, indicating that the Classic algorithm's performance fluctuates greatly for different requests, something which results from its ignorance of the underlying network topology. Another interesting observation is that all the experiments have similar low 5th percentile latencies. These are caused by retrieval requests for files already cached locally on the physical site, and for files cached on a fast host and found after very few routing attempts. This indicates that the Classic algorithm and the NG algorithm perform equivalently for these kinds of requests.

Chord Experiments. Chord provides a decentralized and symmetric peer-to-peer distributed lookup service that can be easily adapted to a file-sharing service. While the functionality provided by Chord is similar to that of Freenet, the two projects have different goals. Freenet is targeted at data anonymity, while Chord is targeted at efficient lookup. Freenet does not assign responsibility for data to specific servers. Its lookups take the form of searches for cached copies, which effectively limits the possibility of providing lower bounds on retrieval latency. In contrast, Chord does not provide anonymity, but its lookup operation runs in predictable time.

To determine the difference in performance between the two systems, and to evaluate Chord's scalability, we performed the same series of experiments as those we performed on Freenet using the same set of workloads. However, we had to replace up to 17 percent of the PlanetLab hosts, since some of those used in our Freenet experiments were unavailable when we performed the Chord experiments; this is the reality of using a wide-area, public testbed. Nevertheless, the results should be comparable, since we carefully replaced the hosts with those having similar geographical separation as the original ones. The results are shown in Figure 9.

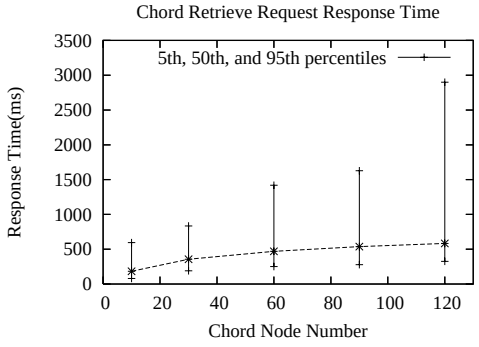

Figure 9: Chord Retrieve Latency

Figure 9 shows the median, the 5th, and the 95th percentile of retrieve latency for Chord. The retrieve latencies are an order of magnitude less for Chord than for Freenet. For Chord, the median latency increases from 183ms to 583ms. This is more pronounced than that reported by Chord's authors [16]. We believe this is due to a difference in experiment configuration. In particular, although the Chord authors increased the number of Chord instances in their experiments, their testbed consisted of only ten hosts; experiments with more than 10 instances were conducted by running multiple independent copies of the Chord software at

each site. In our experiments, we increased the geographical scale of the testbed as well as its node count. In that sense, our experiments are more realistic than those of Chord's authors, and the results likely to be more reliable.

Discussion. The experience of deploying and executing experiments on a wide-area testbed demonstrated Weevil's ability to help in scaling up experiments. Although we had no prior experience with Freenet and Chord, we were able to set up the experiments within several hours. The primary difficulties introduced by PlanetLab are the high network latency and instability of the testbed hosts, which led us to improve the efficiency and robustness of Weevil's master control script.

To begin to quantify our experience using Weevil on PlanetLab, we collected five metrics for the Chord experiments, as shown in Table 1. *Setup time* is the time it takes Weevil to perform the workload partitioning and script generation (actions 1-3 in Figure 3). *Deployment time* is the time it takes Weevil to deploy the actor workloads and the generated scripts to the hosts, plus the time it takes to start the component software on each host and to decide the time at which all the actors should begin processing their workloads (action 4). *Configuration differences* is the number of lines of the experiment configuration file that are changed. *Script differences* is the number of different lines in the generated scripts. For each column in the table for these two metrics, the value given is relative to the experiment represented in the previous column. *Generated files* is the number of files created by Weevil for the experiment.

Scale	10	30	60	90	120
Avg. setup time (sec)	2	7	24	58	114
Avg. deploy. time (sec)	11	20	24	38	59
Config. differences	–	31	41	41	41
Script differences	–	758	1148	1178	1208
Generated files	41	121	241	361	481

Table 1: Experiment Scale Modification

These numbers clearly illustrate that Weevil allows the experiment to be easily and quickly reconfigured to handle changes, that generation and automation are effective ways to configure and manage large-scale experiments, and through parallel communication and file transfer, the overall deployment time scales well. Between these experiments, we made changes simultaneously in several dimensions, including the number of hosts, components, and actors. A more systematic evaluation of Weevil's ability to leverage automation, with changes made in each dimension independently, is discussed below.

5.4 Automation

We now present a quantitative evaluation of the benefits of Weevil's automation features. In particular, we measured the effort involved in switching between different experiments with Chord performed on PlanetLab. We considered varying degrees of modification, from minor tweaks to substantial changes. Specifically, we considered testbed modifications, parameter modifications, SUE modifications, and workload modifications. We collected the same five metrics as those in Table 1 for each series of these experiments.

Testbed Modification. When experimenting with distributed systems it is common practice to tune and tweak an experiment on a small testbed and then progressively ramp up the size of the testbed as the experimental setup becomes more solid. Other changes to the testbed might be driven by the need to change or replace a machine, or to migrate an experiment from a local-area testbed to a wide-area testbed. Since testbed changes are likely to be common, we wanted to see how Weevil would cope with them.

We experimented with a Chord network consisting of 120 components, each with its own actor. The initial experiment was deployed on a testbed with 10 host machines; subsequent experiments were conducted with 30, 60, 90, and finally 120 hosts. The bulk of the configuration files were shared verbatim between the experiments except three parts. First, host descriptions for the new machines were added to the testbed description. Next, the existing component/host mapping was adjusted to evenly spread the components across the larger testbeds. Finally, we adjusted a few component properties that are affected by the modified component/host mapping. These changes are made concisely by using Weevil's programmability. In the example below, we updated the component/host mapping from the 10-machine experiment to the 30-machine experiment by changing a single value:

```
< WVL_SYS_Foreach(‘i’,
< ‘WVL_SYS_ComponentHost(‘CHMapN’i, ‘N’i, ‘H’eval(i%10))’,
< WVL_SYS_Range(‘’, 0, 119))dnl
----
> WVL_SYS_Foreach(‘i’,
> ‘WVL_SYS_ComponentHost(‘CHMapN’i, ‘N’i, ‘H’eval(i%30))’,
> WVL_SYS_Range(‘’, 0, 119))dnl
```

With these simple configuration changes, Weevil was not only able to automatically deploy the system onto the larger testbeds, but also automatically relocate each actor to its corresponding component's new host. The metric values of these experiments are shown in Table 2.

Number of hosts	10	30	60	90	120
Avg. setup time (sec)	113	113	113	114	114
Avg. deploy. time (sec)	51	40	43	53	59
Configuration differences	–	23	33	33	33
Script differences	–	804	664	664	514
Generated files	481	481	481	481	481

Table 2: Testbed Modification

The numbers in the second row of Table 2 highlight Weevil's parallel file transfer capabilities, as the deployment time grows slowly with the number of hosts. The 10-host experiment has a longer deployment time than the 30- and 60-host ones because there is less opportunity for parallelism with so many components residing on each host. The most important result of this experiment is the amplification achieved by Weevil's generative capabilities. Small changes in the configuration were multiplied by a factor of up to 35 in the resulting generated scripts. This amplification, coupled with Weevil's consistently quick setup time, clearly demonstrates the benefits of automation and generation for the task of modifying the testbed. The number of files generated depends on the number of components and the number of actors, which were fixed for this experiment.

System Parameter Modification. An engineer usually evaluates a system under different configurations. We per-

formed two experiments with different parameter settings to characterize Weevil's ability to support this. In the first experiment, we changed the value of an existing parameter, and in the second case we added a new parameter in order to override its default value. Normally, an engineer would modify such parameters directly in the start command or in the configuration file of each component. However, since Weevil supports parameterization of start scripts and configuration files, we were able to configure each component type centrally in our experiment configuration in a clean and flexible way. When specifying a new parameter, we updated both the start script and each component's properties. For example, we wished to create three virtual nodes on each Chord node, which can be accomplished with the *-v* option in Chord's start command. We did not include this option in our initial start script because we used the default value "1" in that experiment. We were able to accomplish this simply by adding the following option to the start script

```
-v 'WVL_Component_'WVL_Component_ID'_vn'
```

and by adding the following component property declaration to the experiment configuration file

```
> WVL_SYS_Foreach('i',
> 'WVL_SYS_ComponentProp('N'i, 'vn', 3)',
> WVL_SYS_Range('', 0, 119))dnl
```

The metric values of these experiments are shown in Table 3. Once again, the data show how a few tweaks in the system configuration result in a large number of changes to the management scripts.

Parameter	Initial	Updated	New
Avg. setup time (sec)	114	114	114
Avg. deploy. time (sec)	59	59	59
Configuration differences	–	1	2
Script differences	–	120	120
Generated files	481	481	481

Table 3: Parameter Modification

SUE Modification. The preceding experiments examined adjustments or other minor changes to an experiment. In this example, we explore a more fundamental change involving the configuration of the SUE itself.

Number of components	10	30	60	90	120
Avg. setup time (sec)	34	39	52	75	114
Avg. deploy. time (sec)	42	32	41	52	59
Configuration differences	–	7	7	7	7
Script differences	–	846	1056	1026	906
Generated files	151	211	301	391	481

Table 4: SUE Modification

We altered the SUE of an existing experiment by increasing the number of components from 10 to 120, with the number of hosts and actors fixed at 120. To accomplish this we instantiated the new components and added their properties and relations. We also updated the component/host mapping to accommodate the new components. Again, all of these can be accomplished in just several lines utilizing Weevil's programmability. The metric values for these experiments are shown in Table 4. As the difference numbers show, even fundamental changes in an experiment can be accomplished with very small configuration changes, which are amplified by Weevil as necessary when generating files.

Workload Modification. Load testing is very common for distributed systems. One way to modify the load on a system is by changing the number of actors. In our experiments, a Chord network of 120 nodes was deployed on 120 PlanetLab hosts. We experimented with this by increasing the number of actors from 10 to 120. In these experiments, each actor is mapped to an individual Chord node. One wrinkle with this experiment is that adding more actors requires the creation of new workloads with correspondingly more independent processes. These new processes are mapped to existing components through the actors in the experiment configuration file.

Number of actors	10	30	60	90	120
Avg. setup time (sec)	71	75	84	98	114
Avg. deploy. time (sec)	34	49	53	55	59
Configuration differences	–	5	5	5	5
Script differences	–	183	273	273	273
Generated files	371	391	421	451	481

Table 5: Workload Modification

The data for these experiments, shown in Table 5, confirm what the previous experiments showed, namely that Weevil allows the experiment to be easily reconfigured to handle both shallow and fundamental changes, and that generation and automation save a lot of effort on the part of the engineer, making it easier to conduct iterative experiments.

6. RELATED WORK

In the context of testing distributed systems, a number of tools (e.g., DECALS [9], RiOT [7], and TestZilla[4]) have been targeted at large-scale distributed applications. However, they are essentially control and logging systems for executing user-provided test scripts in parallel. Unlike Weevil, none of them consider automated workload generation and script construction and, hence, only provide a service that operates at a much lower level. Additionally, none of the documentation available for these tools report evaluation of their operation on a wide-area testbed.

Users of PlanetLab have contributed a number of tools and services to streamline its usage. The most popular of these tools, Parallel SSH [4], vxargs,[5] plDist,[6] and CoDeploy,[7] are simply intended to provide efficient deployment files or execute parallel commands. Two others, the Nixes Tool Set[8] and the PlanetLab Application Manager,[9] are mainly focused on distributed service deployment and maintenance. None of the tools assist the engineer in systematic, repeated experimentation.

Another widely used testbed, Netbed/Emulab [19], provides an integrated emulation and simulation environment, where traffic-shaping techniques are used to configure a cluster to a desired profile. While Netbed provides support for system deployment and experiment management, there is, similar to PlanetLab, no support for workload and customized script generation, and automatic collection of application-level data is not supported.

[4]http://www.cs.cornell.edu/vogels/TestZilla/default.htm
[5]http://dharma.cis.upenn.edu/planetlab/vxargs/
[6]http://www.arl.wustl.edu/~mgeorg/plDist.html
[7]http://codeen.cs.princeton.edu/codeploy/
[8]http://www.aqualab.cs.northwestern.edu/nixes.html
[9]http://appmanager.berkeley.intel-research.net/

A few other distributed system testing frameworks employ a model-driven approach to configuration, but their frameworks and models are fairly narrow in scope. Two examples are the JXTA Distributed Framework[10] and CCMPerf [10], which is targeted at CORBA applications.

Finally, Skoll [13] supports "distributed quality assurance (QA)", whereby QA activities are divided into subtasks and distributed onto the computing resources of worldwide user communities. This approach allows applications with a large space of possible configurations to be tested in parallel. To our knowledge, Skoll has not yet been applied to distributed systems.

7. CONCLUSION

In this paper we have presented Weevil, a framework for automating experiments on distributed testbeds. Weevil is targeted at highly distributed systems, removing many practical obstacles, such as scale and heterogeneity, that hinder experimentation. The framework covers workload generation, as well as experiment deployment and execution. Our simulation-based workload generation approach offers a complementary means to generate synthetic workloads, while our model-based configuration and automated script construction provide a higher-level service than other available tools.

To date we have used Weevil on a broad set of distributed systems drawn from three common operating paradigms: event-based, peer-to-peer, and traditional client/server. Further, we have used Weevil to perform experiments on both local-area and wide-area testbeds. Our workload generation approach allowed us to create workloads for real scenarios and to successfully reproduce and broaden previously published experimental results. Finally, we have evaluated Weevil's ability to leverage automation across multiple experiments. All of these experiences demonstrate that Weevil is readily configured for different systems, is capable of managing large-scale experiments, and is a genuinely efficient, labor-saving experimentation tool.

Our immediate plans for Weevil are to include support for dynamic workloads. Often, an experiment requires an actor to tailor its behavior based on output obtained from the SUE during execution. To support this, Weevil must provide a means for the user to specify rules for workload modification, to set up response/request relations, and to make the workload responsive to experiment execution.

8. REFERENCES

[1] R. Adams. Take command: The m4 macro package. *Linux J.*, 2002(96):6, Apr. 2002.

[2] M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Transactions on Software Engineering*, 29(12):1059–1071, Dec. 2003.

[3] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[4] B. Chun. *pssh HOWTO*. Intel Research Berkeley, Nov. 2003.

[5] I. Clarke. *Freenet's Next Generation Routing Protocol*. Freenet Project, July 2003.

[6] I. Clarke, S. G. Miller, T. W. Hong, O. Sandberg, and B. Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[7] S. Ghosh, N. Bawa, G. Craig, and K. Kalgaonkar. A test management and software visualization framework for heterogeneous distributed applications. In *Proceedings of the 6th IEEE International Symposium on High Assurance Systems Engineering (HASE '01)*, pages 106–116, Boca Raton, Florida, Oct. 2001.

[8] D. Guerrero. System administration: Caching the web. *Linux J.*, 1999(58es):11, 1999.

[9] A. Hubbard, C. M. Woodside, and C. Schramm. Decals: Distributed experiment control and logging system. In *Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research*, page 32, Toronto, Ontario, Canada, Nov. 1995.

[10] A. S. Krishna, N. Wang, B. Natarajan, A. Gokhale, D. C. Schmidt, and G. Thaker. CCMPerf: A benchmarking tool for CORBA component model implementations. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, pages 140–147, Toronto, Canada, May 2004.

[11] B. Laurie and P. Laurie. *Apache: The Definitive Guide*. O'Reilly and Associates, 3 edition, 2002.

[12] A. Martinez, Y. Dimitriadis, and P. de la Fuente. Towards an XML-based model for the representation of collaborative action. In *Proceedings of the Conference on Computer Support for Collaborative Learning (CSCL '03)*, pages 14–18, Bergen, Norway, June 2003.

[13] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering (ICSE '04)*, pages 459–468, Edinburgh, United Kingdom, May 2004.

[14] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. *ACM SIGCOMM Computer Communication Review*, 33(1):59–64, 2003.

[15] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the Australian UNIX and Open Systems User Group Conference (AUUG '97)*, pages 243–255, Brisbane, Australia, Sept. 1997.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '01)*, pages 149–160, San Diego, California, Aug. 2001.

[17] L. Tauscher and S. Greenberg. How people revisit web pages: Empirical findings and implications for the design of history systems. *International Journal on Human-Computer Studies*, 47(1):97–138, 1997.

[18] Y. Wang, M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Weevil: A tool to automate experimentation with distributed systems. Technical Report CU-CS-980-04, Department of Computer Science, University of Colorado, Oct. 2004.

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

[20] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 16–31, Kiawah Island, SC, Dec. 1999.

[10] http://jdf.jxta.org