Weevil: a Tool to Automate Experimentation With Distributed Systems

Yanyan Wang Matthew J. Rutherford Antonio Carzaniga Alexander L. Wolf

Software Engineering Research Laboratory
Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430 USA
{ywang,rutherfo,carzanig,alw}@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-980-04
October 2004

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 two key elements of the experimentation activity: (1) workload generation and (2) experiment deployment and execution. Our approach is founded on a suite of models that characterize the client behaviors that drive the experiments, the distributed system under experimentation, and the testbeds upon which the experiments are to be carried out. The models are used by simulation-based and generative techniques to automate the 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 on a variety of distributed systems. Our experience shows that this framework can be readily applied to different kinds of distributed system architectures, and that using it for meaningful experimentation is advantageous.

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 contrast to the 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 general-purpose 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, repeatable 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.

The work described here focuses on this latter aspect. Specifically, we propose a comprehensive experimental framework that helps to manage and automate experiments with distributed systems. It covers a simple two-phase experimentation process.

1. *Workload generation:* A distributed system workload is a time-ordered list of service calls from clients to system access points. While there are many ways to create such workloads, they are still not flexible enough and unavailable for some scenarios. Our approach to workload generation is to perform an offline simulation of expected client behavior and record the resulting service calls. During an experiment run, these service calls are issued as inputs into the subject system.

2. *Experiment deployment and execution:* This activity applies a workload to a system and gathers runtime data. To achieve this, the subject system, the workload, and the framework codes are first deployed across an experimentation testbed and then configured. Next, the system is started and stimulated by the execution of service calls at the times and locations dictated by the workload. When the experiment terminates, output and diagnostic data are collected from across the testbed to facilitate efficient data analysis and reduction.

To support this process, we have developed an experimentation framework, called Weevil, consisting of a synthetic-workload generator coupled with an environment for managing the deployment and execution of experiments. Weevil is intended to facilitate experimentation activities for distributed systems by providing engineers with a flexible, configurable, automated and, thus, repeatable process for evaluating their software on a networked testbed. Further, Weevil is not targeted at a specific system or application model, but rather it is a generic, programmable tool.

The novelty of our workload generator is that it is powered by a discrete-event simulation engine. This enables a developer to *define* a workload by programming client behaviors within the simulator. The workload is then *generated* by running the simulation to produce the list of service-call times and locations that embody the workload. This simulation-based approach scales well with the number of clients.

Given a workload, Weevil applies it to a specified configuration of the subject system. To support the widest variety of subject systems and testbeds, our goal in this phase is to achieve complete automation while maintaining configurability and adaptability of the framework. Our approach, therefore, is to support virtually all distributed systems through the same generic model, and to make minimal assumptions about the services provided by testbed nodes (i.e. we only require user-level remote shell access). The primary contributions of Weevil in this phase are a set of conceptual models for a distributed system experiment. Using the models, an engineer can concisely describe both the subject system and the testbed involved in an experiment. Weevil deals with heterogeneity by generating customized controls from these models that automate the experiment deployment and execution process. This is in contrast to most related systems that require software engineers to provide this functionality themselves.

We have experimented with numerous client behavioral models, and a variety of different distributed systems. Our early experience, reported here, demonstrates that Weevil facilitates the generation of useful workloads for complex, interdependent client behavioral models, and that those workloads can be readily applied to different software systems over large testbeds with a high degree of automation. We have also used Weevil to successfully duplicate conclusions drawn by researchers from actual system execution, thereby demonstrating its applicability to real world problems.

This paper is structured as follows: in Section 2 we introduce our framework; then we detail workload generation phase and experiment deployment and execution phase in Section 3 and Section 4 respectively; in Section 5 we report our experience in using this framework; we survey related work in Section 6, and in Section 7 we conclude and discuss ways to complete and extend this research.

2 Weevil

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 actuates the SUE as dictated by the client's workload. An experiment can be understood as a two-phase process in which (1) a workload is generated for the actors, (2) the SUE is deployed on the testbed with the actors actuating it based on the workload and the results are collected afterward. Weevil supports the two-phase process through a central-controller architecture in which a master script controls the whole process. Workload generation is conducted on the master's host as presented in Section 3. In the experiment deployment and execution phase (described in Section 4) the master generates all the controls, deploys them together with the workload and the SUE on the testbed, coordinates all the system components and the actors to execute the experiment, and gathers data after the experiment terminates.

Throughout the following sections, we refer to an example experiment aimed at studying the performance of web proxies.

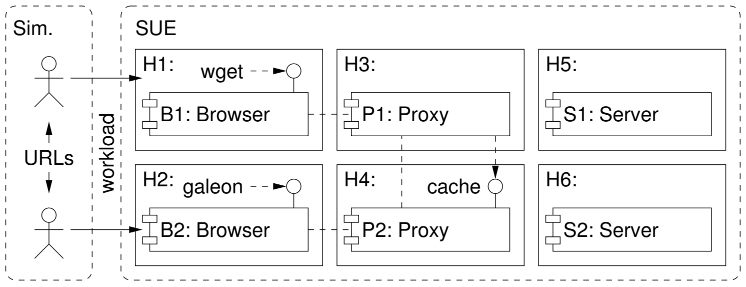

Figure 1: Deployment Diagram of Web-Proxy Experiment

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 performance 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. In this example, the client programs (i.e. the web browsers) are included in the SUE, which is simply a modeling decision.

3 Workload Generation

Weevil supports generation of synthetic workloads in a way that is quite different from existing workload generators. A common practice (discussed further in Section 6) is to generate a workload on the basis of a statistical model that abstracts usage patterns of the SUE. For the web-proxy experiment of Figure 1, a workload generator would be implemented by generating per-actor lists of URLs chosen at random from a given set of URLs following a Zipf distribution, which represents the popularity of each URL. 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.

In contrast, we take an operational approach that models user behavior directly. Details for operational behavior models could come from, for example, empirical traces [25], detailed user behavior profiles [20], or an experiment 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.

3.1 Simulation-Based Workload Generation

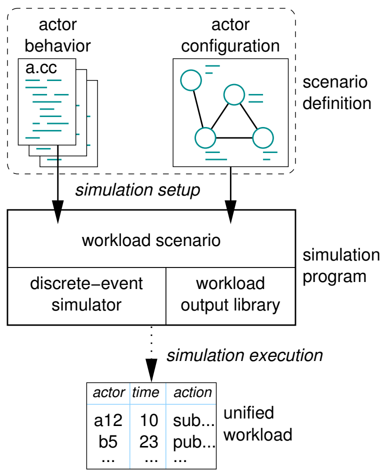

Figure 2: Simulation-Based Workload Generation

Weevil's simulation-based workload generation process is illustrated in Figure 2. It allows the engineer to program one or more types of actor behaviors. These behaviors are programmed in a common programming language (e.g., C++) with the support of Weevil's workload-generation library, and may therefore execute arbitrary functions and maintain arbitrary state. After programming the actor behavior types, the engineer can populate a scenario consisting of many 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 simulation library and then 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 must be applied to the SUE during experiment execution.

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.[1] The library supports message communication between `Process`es, 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 the 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. Examples of actor behavior implementations are presented in Section 3.3.

3.2 Design Rationale

Our motivations for designing and using a simulation-based workload generator are its inherent flexibility and scalability. It is flexible in two dimensions. First, it is versatile in the sense that it can be immediately used to program workload generators based on statistical models. In fact, those generators reduce to scenarios with independent stochastic

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

`WeevilProcess`es. 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.

3.3 Examples

In order to demonstrate the flexibility of Weevil's workload generator, we have used it to produce workloads corresponding to a variety of statistical and operational actor models.

Statistical actor models are not the primary focus of our workload generator. Nonetheless, our experience shows that Weevil, in combination with a scientific library,[2] can easily support these models. The more interesting workloads that Weevil can produce are those derived from operational behavioral models, and, in particular, those expressing *interdependent* behaviors.

The following examples show the usages of two types of `WeevilProcess`, either a *sequential process*, or a *reactive process*. A sequential process is defined simply by programming the `main` method, which defines the body of the process directly. A process can use the `signal_event` method to signal other processes, and the `wait_for_event` method to receive events signaled from other processes. In contrast, a reactive process is defined by programming the `process_event` and `process_timeout` methods, which define the functions executed by the process in response to an event signaled from another process and a timeout, respectively. It is used when the process may have other reactive behaviors in parallel to its routine behaviors.

3.3.1 Intra-Actor Interdependent Behavior

One type of interdependent behavior is one where future requests need to incorporate data from a previous request. As an example, consider a distributed publish/subscribe system where actors use an access point to publish notifications and subscribe for notifications of interest. A common behavior for a subscriber actor could be to periodically subscribe and unsubscribe, with every unsubscription matching the previous subscription. This behavior can be programmed as follows:

```
class Subscriber : public weevil::WeevilProcess {
public:
  virtual void main() {
    string s;
    for(int i = 0; i<10; ++i) {
      Sim::set_timeout(random_interval());
      Sim::wait_for_event();       // here we sleep a bit
      s = random_subscription();
      workload_output("subscribe(" + s + ")");
      Sim::set_timeout(random_interval());
      Sim::wait_for_event();       // we sleep again
      workload_output("unsubscribe(" + s + ")");
    }
  }
};
```

In this case, the behavior is represented as a class and is defined simply by programming the `main` method of that class. As in the previous example, Weevil can easily generate workloads for various types of session-oriented protocols, where actors maintain a virtual connection with the system by providing an immutable session identifier throughout their interaction with the system.

[2]http://www.gnu.org/software/gsl/

3.3.2 Inter-Actor Interdependent Behavior

Another type of interdependent behavior is one where actors communicate and coordinate their interactions with the SUE. To exemplify and test this kind of behavior, we have created an emergent scenario that simulates the propagation of the Code Red worm through a network. The generated traces of infections could be used, for example, as a workload to evaluate a distributed intrusion detection system.

Our scenario uses a simplified version of the Code Red worm.[29] The actual Code Red worm generates 100 threads. Each of the first 99 "probe threads" iterates through a set of randomly chosen IP addresses, trying to establish a connection to port 80 on that address. If the target is a web server and the connection is successful, the worm attempts to break into the server and install a copy of itself. In our example, we consider a simplified scenario with the same algorithm but only one probe thread.

To generate a workload, we consider each host instance as a workload process `Host`, the infection as the only action of that process, the probing of each host is represented by a `Probe` object. The configuration of each workload process consists of a boolean flag to determine if the represented host is susceptible to probes, a boolean flag to determine if it has been infected, the number and time distribution of its probes, and a vector of neighbors it would probably probe. A host that is already infected or unsusceptible will simply ignore probe signals. Otherwise, a probe will output an "INFECT" action to the workload, and will initialize the target host to begin its probe actions.

The code to implement this operational model is shown at the top of Figure 3.

```
class Probe : public Event { };

class Host : public weevil::WeevilProcess {
  bool susceptible, infected;
  unsigned int num_probes;
  int probe_interval;
  vector<neighborhood*> neighbors;
public:
  virtual void init() {
    if (infected && (num_probes > 0)) {
      num_probes--;
      Sim::set_timeout(probe_interval);
    }
  }
  virtual void process_event(const Event* e) {
    if (susceptible && !infected) {
      infected = true;
      workload_output("INFECT");
      init();           // begin probe actions
    }
  }
  virtual void process_timeout() {
    int fi = rand_normal_range(0, neighbors.size()+1);
    signal_event(neighbors[fi]->id, new Probe(),
                 neighbors[fi]->signaldelay);
    if (num_probes > 0) {
      num_probes--;
      set_timeout(probe_interval);
    }
  }
};
```

Figure 3: Programmed Worm Behavior

Notice that in this case, the behavior is defined as a reactive process by implementing the `process_event` and `process_timeout` methods. The `Probe` class represents an event signaled to a simulation process. Notice also that the `Host` class defines the `init` method, which can initialize workload processes before the simulation starts.

The scenario consists of 100 `Host` processes, 59 of which are web servers listening on port 80 (i.e., with `Host::susceptible = true`). Two web servers are initially infected (with `Host::infected = true`). The result of this worm propagation is shown in Figure 4.

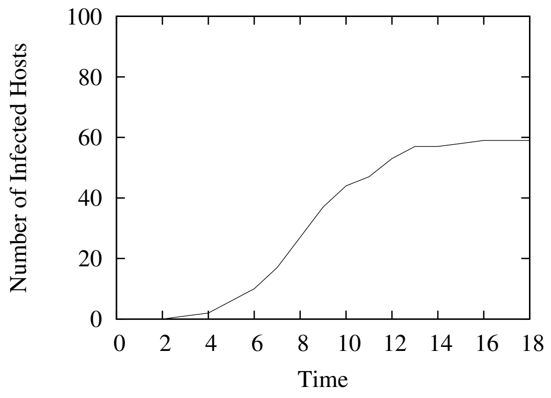

Figure 4: Simulated Worm Propagation

4 Experiment Deployment and Execution

Weevil directly supports experiment deployment and execution. The overall process is depicted in Figure 5. 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 data generated by Weevil or by the SUE during experiment. Solid arrows represent normal input/output data flow, whereas dotted arrows represent the execution of scripts.

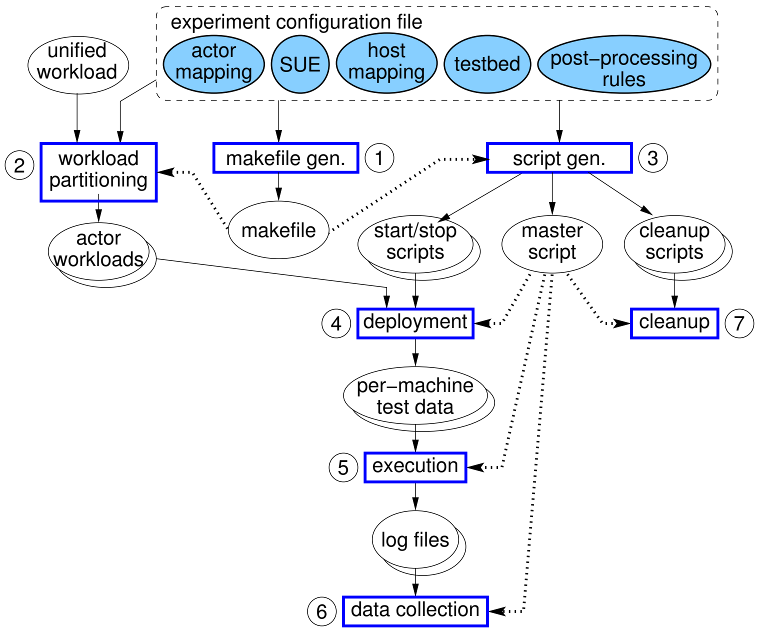

Figure 5: Weevil Experimentation Process

We have taken an automated, model-based approach to Weevil's configuration 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, instead, they only deal directly with a relatively concise set of configuration parameters; (2) models can be shared between experiments and easily tweaked when experiments must be changed;

and (3) Weevil's generative capabilities transparently handle much of the complexity brought about by systems' scale and heterogeneity.

4.1 Experiment Configurations

For each experiment, Weevil must be provided with a workload and experiment configurations of 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 configurations are represented by the dark ovals along the top of Figure 5. They are programmed in GNU m4[2] by calling Weevil-defined macros to instantiate the model elements. 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 it encounters.

4.1.1 SUE Model

The conceptual model of an SUE is shown in Figure 6. As this figure shows, an SUE is comprised of typed *Components*

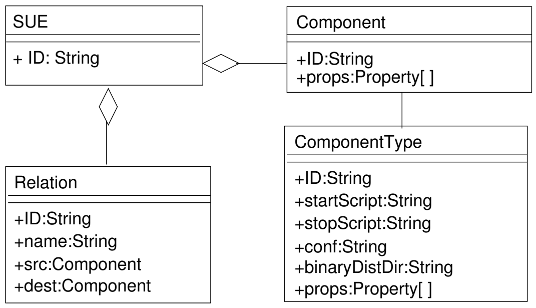

Figure 6: SUE Conceptual Model

and *Relations* between them. Each *ComponentType* instance has *startScript* and *stopScript* attributes, and optionally a *conf* attribute that contains the contents of a configuration file. *ComponentType* also has an attribute, *binaryDistDir*, describing a location to serve as a repository for the binary package. Both *Component* and *ComponentType* allow system-specific properties to be assigned to them. These properties can be referenced in the start and stop scripts and configuration file and are resolved during script generation using m4's macro expansion. The combination of start and stop scripts and the configuration file provided sufficient expressiveness to configure all systems with which we experimented.

The *Relations* contained in an SUE model are used to represent any binary associations between components. These are 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 about another component during generation.

4.1.2 Testbed Model

Weevil makes minimal assumptions about the testbed. It only requires user-level remote shell access. As shown in Figure 7, a *Testbed* has an identifier and a collection of *Host*s. *Host*s have a number of attributes providing local paths to various programs needed by Weevil. A *Host* also has an optional list of properties that can be used to contain any system-specific information that must be known for each host. The *hostType* attribute is used to partition the hosts into categories that each have their own binary package.

4.1.3 Mappings

The two models described above and the workload are largely independent. This independence means that they can be composed together in many different combinations by providing two mappings between them. The first mapping

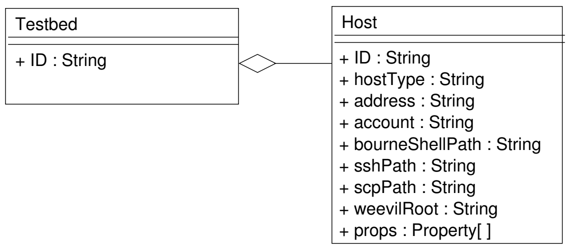

Figure 7: Testbed Conceptual Model

simply associates each *Component* in the SUE with a *Host* in the testbed. Weevil is quite flexible in this regard since a single host can serve multiple components. The second mapping associates each workload process with a component through an actor. All workload process must be represented by a single component, but any number of workload processes can be associated with a particular component through actors. An actor is implemented as a system-specific program that understands how to actuate the SUE as dictated by the workload. Weevil provides a library to support its implementation in Java or in shell script. In Figure 1, the two different actors use `wget` and `galeon` to turn the URLs contained in the workload into HTTP requests.

4.1.4 Post-Processing Rules

Weevil allows for post-processing of experiment output in two ways: the engineer can specify a list of files for Weevil to copy from each component's and each actor's workspace to the master, or the engineer can provide a script to be executed for each component and each actor after the experiment has completed. The output from this script is copied back to the master.

4.2 Setup and Script Generation

The goal of the setup phase is to "compile" the experiment configurations into the control scripts that will be used for experiment deployment and execution. Initially, the configurations are checked for consistency, and a per-experiment Makefile is generated to control the rest of the process, which consists of actions 2 and 3 in Figure 5.

Action 2 tailors the workload based on the experiment configurations and partitions the unified workload to per-actor workloads. For instance, in the web-proxy example, the network addresses of the web server components are dictated by the component-to-host mapping, and are not generally known when the workload is generated.

Following that, in action 3, three tailored scripts, a start script, a stop script, and a cleanup script, together with a tailored configuration file are generated for each component in the SUE by applying macro expansion to the contents of the relevant attributes of *ComponentType*. Additionally, a single master control script is created.

4.3 Deployment and Execution

At this point, execution of the experiment is straightforward. The master control script deploys the SUE, per-actor workloads, actors and scripts to the hosts, starts all the components, and then starts the actors. The actors begin processing their workloads at a predetermined time that is selected by the master script. The master script waits for all the actors to complete processing their workloads and then executes the stop scripts for each of the components. After all of the components have been terminated, log files are either copied back to the master machine, or post-processing scripts are executed, as configured in the post-processing rules.

5 Experience

Our evaluation of Weevil was divided into three main activities, each with a different focus. First, we concentrated on developing representative experiments for five different distributed systems. The goal of this activity was to validate the expressiveness of our conceptual models and their adaptability to real systems. Second, we conducted experiments intended to measure the cost savings achieved by the automation and generative techniques. These experiments covered some typical situations that an engineer might encounter when evaluating a system. Third, we used Weevil to evaluate the cache hit rate in different scenarios of a web cache system, thereby demonstrating Weevil's applicability to real world problems.

5.1 Survey of Case Studies

To evaluate the applicability of Weevil as a whole, we applied it to five different distributed systems: Siena [9], Elvin [24], MobiKit [8], Freenet [10], and a composite web-cache system made up of Squid [15] proxies and Apache web servers.[3]

We chose these systems because they share the following interesting features: all are component-based distributed systems; individual components are configurable through run-time directives, all provide client APIs, and all support federation of components. The five systems provide a representative sampling from the broad spectrum of distributed systems. Each has unique characteristics that requires special consideration when configuring Weevil to perform experiments. Below we illustrate the ways in which Weevil was able to accommodate some of these specific differences.

5.1.1 Siena

Siena implements a publish/subscribe service through a network of Siena servers. Each Siena client accesses the service through a local server. In our experiment, the servers are organized in a hierarchy, using a parent/child relation. Actors running local to each component inject subscription requests and publications into the SUE using Siena's Java API. The hierarchy of Siena servers is established by starting up each server with a command-line parameter that indicates the address of its parent server. This complicates the execution of an experiment because it requires each server start script to adapt to the testbed and the topology. Fortunately, Weevil simplifies this situation by allowing the start scripts to be parameterized by the properties of the component as well as the testbed. We therefore solved this configuration issue as follows:

1. we assigned each Siena server component a "port" property and a "protocol" property;
2. we configured the Siena hierarchy by defining a "parent" *ComponentRelation* between server components; and
3. we used the above macros in the start script defined for the "SienaServer" `ComponentType`, so that for each Siena server, the "parent" parameter of its start script is automatically bound to its parent's testbed machine and server port.

Using this straightforward approach, experiment setup can be easily adapted for different testbeds and different hierarchical topologies.

5.1.2 Elvin

Elvin is another publish/subscribe middleware system that supports federation of servers. For this experiment, we only had access to the Linux distribution of Elvin, but our testbed machines were all running FreeBSD. We were able to work around this issue by using FreeBSD's Linux emulation package. However, due to security restrictions, we were unable to automate the configuration for the Linux emulator and therefore we had to rely on the pre-installed SUE on each testbed host. Weevil accommodates this by simply skipping the deployment of the SUE, as indicated by a blank *binaryDistDir* attribute for the "Elvin" *ComponentType*. In addition to that, we had to configure Weevil's scripts to access Elvin's pre-installed binaries, which was also easily accomplished by setting the execution path for each testbed host.

[3]http://httpd.apache.org

5.1.3 MobiKit

MobiKit is a mobility framework built for distributed publish/subscribe systems. The mobility service is implemented through *mobility proxies*, which are independent, stationary components that run at the access points of a publish/subscribe system. Mobile clients of a publish/subscribe system move through the network attaching to nearby MobiKit proxies, providing transparent access to the publish/subscribe systems.

The most significant problem posed by MobiKit is that it requires each mobile client to know both the proxy and the publish/subscribe server to access. This information is clearly dependent on the configuration of the SUE and of the particular testbed. To solve this additional binding problem, we created a workload that is itself parameterized by macros that are expanded during Weevil's workload partitioning phase.

5.1.4 Freenet

Freenet is a peer-to-peer file-sharing system. An interesting 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's macro expansion to be applied to a configuration file on a per-component basis in a similar way to the component's start/stop scripts.

As shown in Figure 6, Freenet's *ComponentType* contains a *conf* attribute that is the content of its configuration file. This content is retrieved during generation using m4's include mechanism. For Freenet, we added macros into the default configuration file so as to produce portions of it through Weevil's macro expansion. 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’
```

This macro uses the `ListenPort` component property. The template configuration file is expanded separately for each component in the SUE, and the macro `WVL_Component_ID` is defined to the identifier of the component currently being processed. Each component has a `ListenPort` property assigned to it that is stored in a macro of the form: `WVL_Component_<ID>_ListenPort`. So, the macros on the right-hand side of the assignment expand to be the listen port for the component being processed.

5.1.5 Squid/Apache

Squid is a full-featured caching web proxy, which in this experiment serves Apache servers. Binaries for Squid and Apache are available for a number of different platforms. Our testbed makeup required that one of our Squid component instances resided on a Linux machine, while the others were hosted on FreeBSD machines.

This example highlights Weevil's ability to handle heterogeneous testbeds. Weevil supports the deployment of a single component onto a heterogeneous testbed through the *hostType* attribute of each host. When the master script is deploying component software to the hosts, as specified by the component/host mapping, the *hostType* attribute is used to select the appropriate binary distribution directory.

The Squid/Apache experiment also features actors that are based on executable programs rather than specific language bindings and APIs. In particular, we used the `squidclient` program that is part of Squid's distribution to actuate the Squid instances. Weevil is flexible enough to allow the use of any program available on the host machine to process the workload actions. It does this through its shell script actor, which parses the workload files and hands off the execution of the actions to the specified program.

5.2 Leveraging Automation

We now present an initial quantitative evaluation of the benefits of Weevil's automation features. In particular, we measured the effort involved in switching between different experiments. We considered varying degrees of changes, from minor tweaks, to substantial changes. Specifically, we considered testbed modifications, parameter modifications, SUE modifications, and workload modifications.

We collected five metrics for each of these experiments:

Setup time is the time it takes Weevil to perform all the workload and script generation.

Deployment time is the time it takes Weevil to deploy the subject system, the workload, and the framework codes to the hosts, plus the time it takes to start the component software on each host.

Configuration differences is the number of lines of the experiment configuration file that are changed between experiments. Since this is a measure of difference, it is not applicable to the first experiment in each table.

Script differences is the number of different lines in the generated scripts. Again, this metric is not applicable to the first experiment in each table.

Generated files is the number of files created by Weevil for the experiment.

5.2.1 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 on a system consisting of 20 components, each with its own actor. The initial experiment was deployed on a testbed with only three machines, we then augmented this to 10 and then 20 machines. These changes affected only two parts of Weevil's experiment configuration file. First, host descriptions of the new machines were added to the testbed description, and then the existing component/host mapping was adjusted to evenly spread the existing components across the larger testbeds. All other parts of the configuration were shared verbatim between the experiments. With these simple macro changes, Weevil was not only able to automatically deploy the system onto the new machines, but also automatically update references between the components. The results from this experiment are shown in Table 1.

Number of hosts	**3**	**10**	**20**
Setup time (sec)	10	10	11
Deployment time (sec)	11	24	44
Configuration differences	–	19	12
Script differences	–	268	180
Generated files	101	101	101

Table 1: Testbed Modification

One main result from this experiment is that the deployment time increases roughly linearly with the number of hosts. This makes sense since the deployment time is dominated by the communication between the master machine and 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 15 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, which was fixed for this experiment.

5.2.2 System Parameter Modification

An engineer usually evaluates a system under different configurations. To characterize Weevil's ability to support this, we performed a series of experiments with different protocol parameter settings.

We performed two different parameter changes. In the first case we changed an existing protocol parameter specification, and in the second case we added a new parameter specification to change its default value. (These changes were applied to experiments on the Siena system.) 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 commands and configuration files, we were able to configure each component type centrally in our experiment configuration in a cleaner and more flexible manner. Moreover, we utilized Weevil's programmability to assign a global property to all components in one line in the body of a for-loop as shown below:

```
< WVL_SYS_Foreach('i',
< 'WVL_SYS_ComponentProp(i, 'Protocol', 'tcp')', Components)
----
> WVL_SYS_Foreach('i',
> 'WVL_SYS_ComponentProp(i, 'Protocol', 'ka')', Components)
```

When specifying a new parameter, we had to update both the start command and each component's properties (in a loop as above), so there were two lines of the configuration that had to be changed.

The results of this experiment are shown in Table 2. As the data show, through the use of automation and generation Weevil allowed the engineer to perform a few tweaks in the system configuration that resulted in a large number of changes to the management scripts.

Parameter	**Initial**	**Updated**	**New**
Setup time (sec)	11	11	11
Deployment time (sec)	44	44	44
Configuration differences	–	1	2
Script differences	–	59	20
Generated files	101	101	101

Table 2: Parameter Modification

5.2.3 SUE Modification

Some of the more fundamental changes involved significant modifications in the configuration of the SUE.

Number of components	**3**	**10**	**20**
Setup time (sec)	8	9	11
Deploy time (sec)	42	43	44
Configuration differences	–	20	13
Script differences	–	159	170
Generated files	33	61	101

Table 3: SUE Modification

Initially, we altered the SUE of an existing experiment by increasing the number of components from 3 to 10, and then from 10 to 20. To accomplish this we simply instantiated the new components and added their properties and relations. We also updated the component/host mapping to accommodate the new components. The results for this experiment are shown in Table 3. As the difference numbers show, fundamental changes in an experiment can be accomplished with relatively small configuration changes which are amplified by Weevil as necessary when generating files.

5.2.4 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, we increased the number of actors from 3 to 10 and then from 10 to 20 as shown in Table 4. Adding new actors involved creating new workload processes and adding them to the actor configuration. Also, the new workload processes had to be mapped to existing components through new actors.

The data for this experiment 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.

Number of Actors	3	10	20
Setup time (sec)	7	8	11
Deploy time (sec)	40	41	44
Configuration differences	–	7	3
Script differences	–	49	110
Generated files	84	91	101

Table 4: Workload Modification

5.3 Real World Problem

To illustrate how Weevil can actually help provide useful information about a distributed system, we used it to conduct a series of experiments with a cooperative web cache system made up of multiple peer web cache proxies. Specifically, according to a study [27] of traces from the University of Washington and the Microsoft Corporation, cooperative caching improves cache hit rate rapidly with smaller populations, while it unlikely to provide significant benefit for larger populations. The original study was based upon simultaneous traces from multiple proxies at different organizations, data that is usually difficult to obtain.

Based on our understanding of the trace scenarios, we created workloads through simulating behaviors of several actor groups representing the independent organizations in the trace study. The actors within one group tend to have similar interest and the level of intra-group communication is much higher than inter-group communication. All the actors in one group share the same web cache proxy. The proxies for different groups make up a cooperative web cache system. We changed the actor population across a series of experiments and got the results shown in Figure 8.

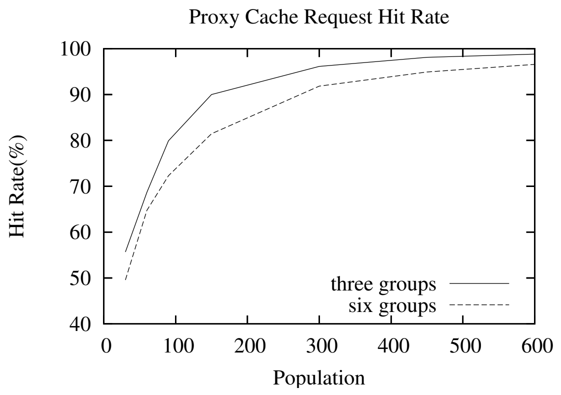

Figure 8: Cache Hit Rate VS. Client Population

Figure 8 shows the hit rates for both experiments with three actor groups and experiments with six actor groups. The former's population is more homogeneous in its Web-access behavior than the latter's. Each of them has an inflection point with a steep increase in request hit rate below and a flatter increase above. We were able to perform these experiments within several hours. The first experiment is the most time consuming to set up, once this is accomplished, the other experiment configurations are simple tweaks of this one.

6 Related Work

There are two main areas of work related to Weevil: workload generation and experimentation facilities.

6.1 Workload Generation

Workload generation has been under study for a long time. In general, three main approaches have been used: trace-based, statistical and operational.

The trace-based approach captures an empirical trace and either subsamples it or permutes the ordering of the requests to generate a new workload different in some respect from the original. This approach is still commonly used in testing tools, such as DieselTest[4], LoadSim[5], and Eggplant[6]. This approach is easy to use and able to generate realistic workloads. However, it has limited flexibility since its generated workload is to a specific environment and it requires the availability of empirical traces.

The statistical approach is currently quite popular. Distributional models are created for the workload factors of interest. Then, data are generated randomly to produce workloads that statistically conform to these models. Many researchers [6, 16, 1, 18, 4, 3] take the resource-oriented approach to directly model the aggregate workload seen by the system. In contrast, the user-oriented approach (Surge [5], Sclient [22]) models each user's activities separately and extracts a set of workload actions from them. The statistical approach's process normally takes too long and requires sophisticated mathematical knowledge [7]. Thus, the normal practice nowadays is to capture a workload based on statistics collected in a completely different situation. This practice affects the realism of the generated workload. Moreover, since the creation of statistical models requires a large number of empirical traces, it is not suitable for modeling emergent behaviors and hypothetical scenarios.

The operational approach uses computational model of each user's behavior to represent the workload characteristics of interest. Compared to the previous two approaches, the operational approach is more flexible and intuitive for engineers since the behaviors are modeled directly. This approach is able to produce workloads for emergent behaviors or for new systems of which very few empirical execution traces are available. Models, including finite state machines and Markov chains, have been adopted [12, 11]. But they are still neither powerful nor flexible enough to model the usage scenarios of distributed systems. In contrast, Weevil's simulation-based workload generation technique takes advantage of imperative languages' expressiveness to provide a way for software engineers to specify very complicated dynamic computational models.

6.2 Experimentation Facilities

As with Weevil, a number of other distributed system experiment tools, such as DECALS [17], RiOT [13], and TestZilla[7], also use the central-controller architecture. However, unlike Weevil, they do not support workload generation or experiment deployment. They also force the engineer to deal with heterogeneity and scalability problem on their own by asking him to provide the workload and all the control scripts. Similar configuration technique has also been used by other testing frameworks, such as JDF[8] and CCMPerf [19]. But they are only applicable to a small range of distributed systems, JDF for JXTA J2SE 2.0 platforms, and CCMPerf for CORBA component models.

Instead of controlling experiments centrally, a system called Skoll [21] separates quality assurance activities into subtasks and distributes them on the computing resources of worldwide user communities. This approach promotes efficiency by allowing many experiments to be conducted in parallel, but it has not yet been applied on distributed system experiments.

Some popular testbeds and their related tools have been developed to provide realistic distributed system experiment environments. PlanetLab [23], a global-scale overlay network, is popularly used to deploy tests of new geographically distributed network services. Its related tools including Parallel SSH[9], the Nixes Tool Set[10], and the PlanetLab Application Manager[11] are mainly focused on experiment deployment and execution, and software maintenance. Another widely used testbed, Netbed/Emulab [26], provides an integrated emulation and simulation environment where

[4]http://www.dieseltest.com
[5]http://www.openware.org/loadsim
[6]http://www.redstonesoftware.com
[7]http://www.cs.cornell.edu/vogels/TestZilla/default.htm
[8]http://jdf.jxta.org
[9]http://www.theether.org/pssh/
[10]http://www.aqualab.cs.northwestern.edu/nixes.html
[11]http://appmanager.berkeley.intel-research.net/

the user can use traffic-shaping techniques to configure the cluster to have the desired profile. While the Netbed management system provides support for software distribution 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. In fact, because of the lack of basic experimentation support from these tools, our next major step is to adapt Weevil to the PlanetLab and Emulab environments.

In practice, prototypes are built early in the design process to give some assurance that the software and hardware infrastructures that they use can provide the required levels of performance. Special experimentation frameworks, such as SoftArch/MTE [14] and LSG [28], were developed to make this possible. Based on user's description about a system's architecture and its usage objectives, a system prototype and its workloads are generated and put into experiment. Although they comprehensively cover every phase of an experiment, they are only designed for the traditional client/server distributed applications constructed from component technologies.

7 Conclusion

In this paper we have presented Weevil, a framework for automating experiments with distributed systems. 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.

To date we have used Weevil on a broad set of five distributed systems drawn from three common operating paradigms: event-based, peer-to-peer, and traditional client/server. We have also evaluated its ability to leverage automation across multiple experiments. We have found that Weevil is readily configured for different systems to provide useful information and is a genuinely efficient, labor-saving experimentation tool.

Our immediate plans for future work are as follows. First, we plan to integrate with different types of testbeds, not only the current real network environments, but also some simulated WANs and overlay-network distributed system testbeds, such as PlantLab. Second, we plan to study extensions of our simulation-based workload generator that would simulate or perhaps emulate the subject software itself. Often, communication among the workload actors is channeled through the subject software, so we could get more realistic workloads by having the subject software involved in that communication.

References

[1] G. Abdulla, E. A. Fox, and M. Abrams. Shared user behavior on the world wide web. In *Proceedings of the second World Conference of WWW, Internet, and Intranet(WebNet '97)*, Toronto, Canada, Oct. 1997.

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

[3] A. Adya, P. Bahl, and L. Qiu. Analyzing the browse patterns of mobile clients. In *SIGCOMM Internet Measurement Workshop*, San Francisco, CA, Nov. 2001.

[4] M. F. Arlitt and C. L. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on networking*, 5(5):631–645, 1997.

[5] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Sigmetrics*, 1998.

[6] M. Busari and C. Williamson. Prowgen: A synthetic workload generation tool for simulation evaluation of web proxy caches. In *IEEE INFOCOM*, 2001.

[7] M. Calzarossa and G. Serazzi. Workload characterization: A survey. *Proceedings of the IEEE*, 81(8):1136–1150, 1993.

[8] 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.

[9] 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.

[10] 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.

[11] M. Deshpande and G. Karypis. Selective markov models for predicting web-page accesses. Technical report, Dept. of Computer Science, University of Minnesota, 2000.

[12] M. R. Ebling and M. Satyanarayanan. Synrgen: An extensible file reference generator. In *Sigmetrics*, 1994.

[13] 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)*, Boca Raton, Florida, Oct. 2001.

[14] J. Grundy, Y. Cai, and A. Liu. Generation of distributed system test-beds from high-level software architecture descriptions. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, 2001.

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

[16] H. Hlavacs and G. Kotsis. Modeling user behavior: A layered approach. In *MASCOTS'99, IEEE Computer Society*, 1999.

[17] 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*, Toronto, Ontario, Canada, 1995.

[18] D. Jagerman, B. Melamed, and W. Willinger. Stochastic modeling of traffic processes. *CRC Press*, 1996.

[19] 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 Real-time Technology and Application Symposium(RTAS '04)*, Toronto, CA, May 2004. IEEE.

[20] 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)*, Bergen, Norway, June 2003.

[21] 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)*, Edinburgh, UK, May 2004.

[22] E. M. Nahum, M.-C. Rosu, S. Seshan, and J. Almeida. The effects of wide-area conditions on WWW server performance. In *Sigmetrics*, 2001.

[23] 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.

[24] 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)*, 1997.

[25] 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.

[26] 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.

[27] 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.

[28] C. M. Woodside and C. Schramm. Scalability and performance experiments using synthetic distributed server systems. *Distributed System Engineering*, 3(1):2, 1996.

[29] C. C. Zou, W. Gong, and D. Towsley. Code red worm propagation modeling and analysis. In *CCS02*. Association for Computer Machinery, Nov. 2002.