Four Enhancements to Automated Distributed System Experimentation Methods*

Yanyan Wang
Dept. of Computer Science
University of Colorado
Boulder, Colorado, USA

Antonio Carzaniga
Faculty of Informatics
University of Lugano
Lugano, Switzerland

Alexander L. Wolf
Department of Computing
Imperial College London
London, UK

ABSTRACT

Experimentation is an essential tool employed by the developers of software systems, especially distributed systems. In prior work we developed a model-driven framework for automating various experimentation tasks, such as workload generation, and demonstrated that it gives the engineer a cost-effective means to conduct large-scale experiments on distributed testbeds. We have enhanced the methods underlying the framework in four significant ways: (1) increasing the *expressiveness* of workloads by allowing for conditional and reactive behaviors; (2) supporting the *repeatability* of experiments through the creation of environment workloads that can control the operational context; (3) enabling the *composability* of application and environment workloads to obtain a broader class of experiments; and (4) extending the *scope* of experiment management to include control over multiple runs. We use the enhancements to conduct a series of interesting new experiments. Specifically, the enhancements allow us to manipulate a fixed-wired testbed so that it simulates a mobile-wireless environment, and to selectively and maliciously inject faults into a system.

Categories and Subject Descriptors

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

General Terms

Experimentation

Keywords

Distributed systems, PlanetLab, Emulab, experiment automation

*Research supported in part by the US National Science Foundation, US Army Research Office, and European Commission under agreement numbers ANI-0240412, DAAD19-01-1-0484, and IST-026955.

ICSE '08, May 10–18, 2008, Leipzig, Germany.

1. INTRODUCTION

Experimentation is an essential tool employed by the developers of software systems. It allows them to gain an understanding of various dynamic system properties and to help tune their systems prior to deployment [14]. Experimentation with *distributed* systems is particularly important and particularly challenging: important in that distributed systems can exhibit a far greater range of behaviors than centralized systems; and challenging in that it can be difficult, costly, and time consuming to reproduce in the laboratory all possible configurations of a distributed system, under all possible usage scenarios, and under all possible environmental conditions or operational contexts.

The terminology of experimentation is not universal. For purposes of this paper we use the following concepts and terms. An individual run of a distributed system is called a *trial* and embodies a particular vector of input values, configuration of components, and mapping of components to network elements. To dampen the effects of unwanted randomness, a series of identical trials is typically executed. We refer to this series of repeated executions of a trial as an *experiment*. A series of different experiments is used to answer an experimental question, such as finding the optimal setting for a system parameter by varying the value of that parameter across the experiments.

At a minimum, experimentation with a distributed system consists of deploying and executing its components on some sort of testbed. But in many cases, the size and degree of distribution of that testbed are crucial to the validity of the experimental results. Recent efforts have led to the development of large, general-purpose testbeds. The best-known example is PlanetLab [13], which is a collection of more than 800 hosts located at over 400 sites around the world that communicate over the live Internet.[1] PlanetLab provides a generic platform on which software systems can be deployed and executed in a realistic multi-host environment. It also has associated with it some tools, such as Plush,[2] for automating deployment and execution. However, PlanetLab does not provide comprehensive support for large-scale experimentation activities.

Experimentation is much more than simple deployment and execution. It involves complex activities such as experimental design, workload generation, data collection, data analysis, and overall experiment management. To address some of the practical challenges in large-scale distributed system experimentation, we developed a framework, called

[1] http://www.planet-lab.org/
[2] http://plush.ucsd.edu/

Weevil,[3] for automating various aspects of the experimentation process. Weevil provides a technique that we refer to as *simulation-based workload generation* for creating the application workloads that embody client usage scenarios. Weevil also makes use of model-driven generative techniques to create scripts that configure, deploy, and manage an individual trial. We have used Weevil to conduct large-scale experiments on PlanetLab, demonstrating the power and versatility of Weevil's workload-generation and modeling features, and documenting the cost savings of automation [18].

Although it proved to be useful, the original version of Weevil had several severe limitations. This paper presents four significant enhancements to our method for large-scale distributed experimentation: (1) increasing the *expressiveness* of workloads by allowing for conditional and reactive behaviors; (2) supporting the *repeatability* of experiments through the creation of environment workloads that can control the operational context; (3) enabling the *composability* of application and environment workloads to obtain a broader class of experiments; and (4) extending the *scope* of experiment management to include automated control over repeated executions of a trial within an experiment.

We describe a series of experiments that demonstrate the impact of the enhancements. The experiments, conducted on a 50-host testbed, examine the performance and robustness of CFS [5], a wide-area, cooperative, replicated storage service built on top of the Chord distributed hash table [16].

Network of mobile hosts: We examine the retrieval response time of CFS when deployed on mobile hosts. To do this, we develop an environment workload that dynamically manipulates the link latencies of the testbed's fixed-wired network to simulate the physical movement of hosts within a wireless network.

Reactive, load-based failures: We examine the robustness of CFS in the presence of a non-random failure model in which the failures are correlated to the load placed on the CFS nodes. To do this, we create an environment workload that injects faults into the Chord network using actions conditionally parameterized by the dynamics of the loads.

Malicious failures: We examine the robustness of CFS in a worst-case, malicious failure scenario in which the failures are caused by an adversary able to anticipate the target of retrieval requests. To do this, we create a parameterized, fault-injecting environment workload that is composed with an application workload.

In the next section we briefly review the basic Weevil framework [18]. We then describe in Section 3 the key ideas underlying the four new enhancements. These ideas, together with the demonstration experiments, are the primary contribution of the paper. We also describe how we extended Weevil to accommodate the enhancements. In Section 4 we detail our experiences using the new facilities to conduct large-scale experiments on the Emulab testbed [19]. Related and future work are discussed in the final two sections.

2. BACKGROUND: WEEVIL

Weevil is a model-driven trial automation framework designed in a generic and programmable way to be used for wide-ranging experimental goals. For example, an engineer might conceive of experiments to reveal interface errors by exercising the basic functionality of a system, to study a system's performance under varying loads, or to tune a system's parameters prior to field deployment. Weevil supports the engineer in carrying out experiments by automating both the creation of application workloads and the management of trials on distributed testbeds.

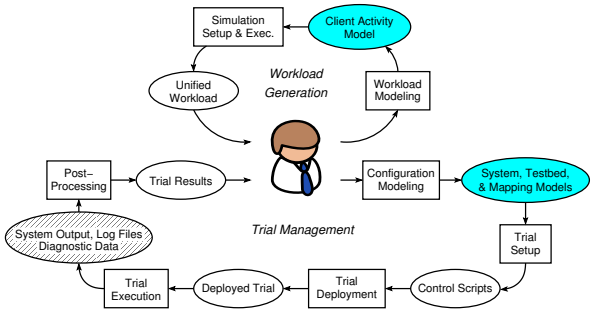

Figure 1: Weevil Automation Process

Weevil embodies a high-level process consisting of the general steps shown in Figure 1. Actions are represented by rectangles. Input and output data for 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 oval represents data generated by the subject system during a trial.

The engineer is responsible for populating several models that capture essential information about a planned trial. There are separate models for the client activity, the subject system, the testbed, the mapping of the subject system's components to the testbed hosts, and the mapping of the clients to the testbed hosts. The first model is used to drive the simulations that generate the application workloads, shown as the top cycle in Figure 1, while the other models are used to generate control scripts for managing the trial, shown as the bottom cycle in Figure 1.

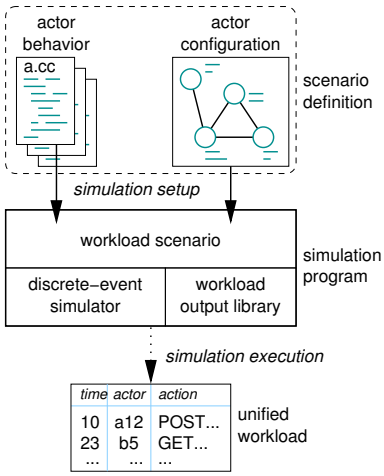

Figure 2: Workload Generation [18]

The input to our simulation-based workload generator is an operational definition of a set of *actors* representing clients, as shown in Figure 2. For example, if the subject of the experiment is a Web caching system, then an actor would represent a person browsing the Web or a robotic

[3]http://www.inf.unisi.ch/carzaniga/weevil/

Web crawler. Actors are instantiated as processes within a discrete-event simulation and those processes are then executed in simulation to reveal the actor behaviors. The output of the simulation is a simple, time-ordered trace of actions performed by the actors. The trace is then processed into one or more workloads.

After the workloads and scripts have been generated, the engineer begins the trial by executing, on a master host, a master control script that in turn deploys, executes, and coordinates several slave control scripts and workloads. The master script is able to automatically redeploy and re-execute slave scripts when a testbed host fails during the trial. Each slave script performs three tasks. First it configures and deploys some portion of the subject system's components. It then initiates execution of those components. Finally, after the trial has concluded, it sends logged output and other diagnostic data back to the master host, and performs any necessary cleanup activities.

During the trial, the subject system is stimulated by the execution of service calls at the times and locations dictated by the application workload. This is implemented by *actor programs*, which are themselves deployed and executed by generated control scripts.

3. KEY IDEAS

In this section we present our four new enhancements to distributed system experimentation methods, and describe how they have been incorporated into the Weevil framework.

3.1 Increased Expressiveness

A workload is intended to provide controlled, repeatable stimuli to the subject of an experiment. A given workload therefore represents a particular usage scenario. In the most general case, the scenario involves distributed actors issuing calls to distributed components of the subject system.

The actors we modeled in our initial approach to experimentation exhibited a restricted class of behaviors. Specifically, they could make service calls on the subject system and communicate with other actors, but they could not react to the dynamic behavior of the subject system, ignoring, for example, any values returned from a service call. This is because the behavior of the subject system is precisely what the experiment is intended to characterize, and therefore is not known in advance and cannot be realistically modeled within the simulation that generates the workload. Although restricted in this way, we could still model interesting behaviors that lead to significant experimental results. An example is the emergent behavior of the Code Red worm propagation, which would be difficult to model faithfully using only analytical means [17].

The benefit of this restricted class of behaviors is that it results in a *stateless* workload, in the sense that the execution of a given workload action does not depend, other than through an intrinsic time ordering, on the execution of some prior workload action. This provides flexibility in the way the workload is deployed. In particular, it can be divided into "subworkloads", one for each of the actors involved in the trial. Moreover, the actor programs can be implemented quite simply, amounting to straightforward interpreters that sequentially read and execute workload actions in turn.

Although we can model many useful actor behaviors under this restriction, it is clear that we are also disallowing many important experiments. For example, we are unable to model how an actor should react to the different values returned by a service call, including the failure of the service call. We must be careful, however, because relaxing all restrictions would require the engineer to write computationally complete actor programs. At that point there would be no practical need for a separate workload-generation phase, simulation based or otherwise, since such programs would be equivalent to the workload generator itself.

Fortunately, we were able to find a way to allow actors to be reactive, and still strike a good balance between expressiveness and simplicity. In particular, we maintain workloads in the general form of a sequence of actions, but allow those actions to be parameterized and/or conditionally executed. The parameters and conditions derive their values from the dynamic state of the subject system and environment through a restricted form of communication.

In Weevil, the workload associated with a particular actor consists of that actor's portion of the global workload plus the actor program that gives the workload an interpretation. The engineer has the freedom to make the actions abstract and the interpretation concrete, or vice versa. Consider, for example, a publish/subscribe communication network as a subject system. The workload may contain generic actions for using virtually any typical publish/subscribe system, while the actor program provides an interpretation of those actions in terms of a particular publish/subscribe API. This would allow the same workload actions to be used to drive comparable experiments on different publish/subscribe subjects, simply by replacing the actor program. Conversely, the actions may contain concrete calls for a particular API, in which case the actor program can simply provide a generic dispatch function. The reusability trade off is clear. But in any case, the primary function of an actor program is to read actions and issue calls during the execution of the trial.

To this basic structure of an actor program we have added a component (typically in a separate thread) that can receive communications and set the values of variables. The purpose of this component is to inform the actor program of events, such as conditions in the environment and/or messages from the subject system, that might influence the workload. The communication component interacts with the interpretation component of the actor program by setting the value of shared variables. These variables can only be used to parameterize an action and/or to populate a conditional that determines whether the action is issued. Correspondingly, we have extended the workload output library so that the engineer can write actor simulation processes that produce the appropriate parameterized and conditional actions.

Figure 3a shows a sketch of an actor simulation process representing a user of the Chord system. The actor's behavior will be conditioned on the value of the variable `compstate`, which is used to indicate whether a Chord access point is available or not. Figure 3b shows a portion of the workload produced by the simulation. Each line results from a call to `workload_output` in the actor simulation process, and includes information about the relative time of the event, the identity of the actor, the service call to be made (a Chord "GET" in all cases shown), a condition, and the variable to which the condition should be applied. Figure 3c shows an actor program, written as a shell script, intended to give an interpretation to the workload actions. In this example, there is only one kind of action, a Chord "GET". A Weevil library function iterates through the workload file,

```
class SimpleChordUser : public weevil::WeevilTProcess {
 // properties initialized according to configuration models
 unsigned int num_files;
 unsigned int num_gets;
 Time avg_delay;

public:
 // actor behavior
 virtual void main();
 ...
};

void SimpleChordUser::main() {
 char action[MAX_ACTION_LEN];
 for (int i = 0; i < num_gets; i++) {
  // schedule the next action, behaving as a Poisson process
  Time current_delay = exponential_rand(avg_delay);
  self_signal_event(NULL, current_delay);
  // wait for the next (timeout) event
  wait_for_event();
  // output a workload action
  unsigned int fid = pick_random_file(1, num_files);
  sprintf(action, "GET(f%d)", fid);
  workload_output(action,"compstate -ne 0","compstate");
 }
}
```

(a)

```
event(251,U77,GET(f784),compstate -ne 0,compstate)
event(297,U78,GET(f1827),compstate -ne 0,compstate)
event(321,U80,GET(f719),compstate -ne 0,compstate)
event(410,U81,GET(f1105),compstate -ne 0,compstate)
event(494,U82,GET(f1159),compstate -ne 0,compstate)
...
```

(b)

```
#!/bin/sh
#first load Weevil's actor program library
. theActorPath/WVL_Experiment_Name"_"WVL_Actor_ID"_lib.sh"
#Weevil defines the configuration variables $comppath and $softwarepath
#as well as the workload variables $action and $parameters
if [ $action == "GET" ]
then
 #assign value to shared variable $compstate
 WVL_AssignVar
 #evaluate the workload condition
 if WVL_CheckCondition
 then
  #issue calls to Chord through its client program filestore
  $softwarepath/filestore $comppath/sock -f $parameters &
 fi
fi
```

(c)

Figure 3: An Actor Process (a), a Sequence of Workload Actions (b), and an Actor Program (c)

calling the actor program. The actor program uses Weevil library functions `WVL_AssignVar`, to set the variable `compstate`, and `WVL_CheckCondition`, to evaluate the condition `compstate -ne 0`. The two functions are implemented as M4 macros that dynamically determine which condition to apply to what variable. The overall effect is to cause the actor program to issue a Chord "GET" service call only when it has access to the service.

In terms of computational power, we have added parameters and conditionals, but not loops. We have also managed to preserve the stateless nature of workloads; the presence of shared variables only has a local effect on workload actions. Thus, we have preserved much of the simplicity of the original form of the workload at the same time as we have increased its expressiveness. In Section 3.3 we discuss how we leverage this simplicity to limit the complexity of workload composition.

3.2 Repeatability

The ability to repeat an experiment is key to the experimental method. The initial version of Weevil supported a certain degree of repeatability through its automation features. It did so essentially by recording application workload and configuration settings in its models, thereby reducing the chance of misconfiguration and procedural errors in individual experiment runs. However, complete repeatability requires full control over every relevant aspect of an experiment. For example, an experiment conducted using the PlanetLab testbed would be repeatable only if the results do not depend on the latency (and other non-functional properties) of the networks and hosts used in the experiment. Obviously, such a basic environmental condition is far from stable in the network—the Internet—underlying PlanetLab [15]. In fact, this natural variability is often a desirable condition, as its realism adds validity to the results. Nonetheless, wherever the operational context is not controllable, experiments are rendered non-repeatable.

In truth, complete repeatability in distributed system experimentation is infeasible. But it can be closely approached if the testbed allows control over the dynamics of the operational context. One example is Emulab [19], which is a large, local-area cluster that offers a generic, multi-host execution environment with a configurable and controllable network layer.[4] Emulab provides several primitive but useful environmental controls, including facilities for traffic generation and shaping, for constructing and modifying a network topology, and for remotely rebooting network hosts. When an engineer wishes to run an experiment using a particular simulated operational context, they configure the routers to act as a network having a given topology, drop rate, congestion, and delay, determine the times at which hosts fail and recover, load the system's components onto the hosts, and then start execution of the components. On the other hand, Emulab suffers from the same shortcoming as PlanetLab: it does not provide the automation features needed to make large-scale experimentation tractable and cost effective.

Thus, to support repeatability, we need to find a means to model, configure, and dynamically alter the operational context, given a testbed that provides an appropriate set of environmental controls. Our approach is based on an extremely simple, yet surprisingly powerful, idea: *treat the testbed environment as just another subject system.* In other words, a controllable testbed is itself a distributed system that can be configured and managed through model-driven generative techniques. Further, and perhaps more importantly, its behavior can be shaped by workloads similar in form to those applied to the subject system. These *environment workloads* embody the dynamics of operational contexts, and their creation can be automated through our simulation-based workload generation technique.

In Weevil we view the environmental controls as elements of an API for controlling the testbed. Against this environment API, the basic Weevil models are populated in the

[4] http://www.emulab.net/

```
event(242,I52,FAIL(successor),,successor)
event(253,I52,JOIN(successor),,successor)
event(439,I49,FAIL(successor),,successor)
event(657,I49,JOIN(successor),,successor)
...
```

Figure 4: Portion of an Environment Workload

same way that we make use of the application API. For example, to generate a workload that mimics a given operational context, *environment actor* simulation processes are written in terms of calls on the environment API.

Although less than ideal, we found that Emulab provides the greatest control of the testbeds available to us. Emulab does this through four highly parameterized programs: `node_reboot`, to reboot one or more hosts; `link_config`, to enable or disable a network link; `tevc`, to modify link bandwidth, delay, and drop probability, as well as to determine traffic rates on a link; and `delay_config`, a second means to modify basic link properties. Notice that `node_reboot` and `link_config` are the admittedly crude means to achieve the effect of causing host failures and changing the network topology, respectively. To supplement these capabilities, we found that we needed to develop some specialized control programs of general utility. For example, we needed to be able to kill a process running on a host. We expect that as distributed testbeds become a more mature technology they will provide such things as primitives.

Figure 4 shows a portion of an environment workload intended for use within a Chord network. This workload injects failures at the specified times and locations. The workload also causes failed nodes to recover and rejoin the Chord network after some time. Notice that the workload is parameterized by the `successor` variable (but a null condition). The value of this variable represents a link in the Chord network, and is assigned dynamically during the experiment. Intuitively, the intent of this workload is to choose which nodes to fail on the basis of the routing algorithm of the Chord network. This scenario is described in greater detail in Section 4.

3.3 Composability

Conceptually, application workloads and environment workloads are independent entities, each representing different phenomena: client usage scenarios on the one hand, and ambient network events on the other. Of course, what an experiment is intended to measure is in part the effect that each has on the other. Concretely, we would also like to have independent application and environment workloads so as to increase their inherent reusability: application workloads on completely different infrastructures (e.g., fixed wired versus mobile wireless) and environment workloads for an infrastructure that can support a variety of different applications.

One way to design an experiment is therefore to create a set of application workloads, a set of environment workloads, and then perform trials that draw from their cross product. But there are also situations where it is important to more finely coordinate the two kinds of workloads.

Having established a common abstraction and structure for application and environment workloads for the purposes of repeatability, we recognize an opportunity to also develop a method for "weaving" or opportunely aligning application and environment workload actions in some non-trivial ways.

```
env   event(242,I77,FAIL(successor),,successor)
      event(251,U77,GET(f784),compstate -ne 0,compstate)
env   event(253,I77,JOIN(successor),,successor)
      event(297,U78,GET(f1827),compstate -ne 0,compstate)
      event(321,U80,GET(f719),compstate -ne 0,compstate)
      event(410,U81,GET(f1105),compstate -ne 0,compstate)
env   event(439,I82,FAIL(successor),,successor)
      event(494,U82,GET(f1159),compstate -ne 0,compstate)
      ...
env   event(657,I82,JOIN(successor),,successor)
      ...
```

Figure 5: Result of a Weaving

Such a composition mechanism would allow us to expand yet further the interesting behaviors that can be easily modeled and automated. For example, together with parameterized and conditional actions, the ability to combine the two kinds of workloads allows us to experiment with various types of malicious attacks and other non-random environmental events and failure models.

Clearly, coordinating the interplay of the subject system and the environment is an arbitrarily complex problem, depending on the nature of the individual behaviors represented by the workloads and on the nature of the desired coordinated behavior. As a first step toward providing such a capability we therefore take a minimalist approach. We assume that a significant and interesting class of coordinated behaviors can be expressed by a simple weaving process in which parameters in one workload can be matched with the parameters in another workload by means of simple textual substitutions and unification. In addition, we take advantage of the sequential, stateless nature of the workloads to give a straightforward semantics to the coordinated behavior: it is a simple merge of the actions from each workload.

Figure 5 shows the result of one possible weaving of the application workload of Figure 3b with the environment workload of Figure 4. Notice that the weaving is such that faults are injected into the successors of nodes to which a "GET" request (from a client application) will soon be routed, resulting in a coordinated behavior mimicking a malicious attacker attempting to cause maximal denial of service.

Because the coordinated behavior is experiment specific, we ask the engineer to define the composition. For example, an engineer evaluating the reliability of a system might want to experiment with a worst-case, perhaps malicious scenario in which host failures are related in space and/or time to the clients that access the system. The engineer would write a program to compose the two sequences of actions from the two workloads, coordinating host failures with client service calls by relating values in one workload with parameters in the other. In practice writing such a program is similar to the interpretation component of an actor program, so there is a good opportunity for code reuse. The resulting sequence of interleaved actions are processed, as before, into workloads associated and deployed with individual application and environment actor programs.

A detailed example of weaving logic is discussed in Section 4.2 and illustrated by Figure 8 appearing in that section.

3.4 Extended Scope

As discussed in Section 3.2, a high degree of repeatability can be achieved through the modeling and control of environmental conditions. This is intended to eliminate un-

wanted sources of randomness. Yet, it is quite natural for experiments on distributed systems to include other stochastic processes (user behavior is one typical case) that might affect the outcome of a trial. To be able to characterize the behavior of the system within a desired confidence interval in such cases, it is necessary to obtain independent measurements and, therefore, necessary to execute multiple trials. The challenge here is to provide a useful means to automate this basic experimentation process, thereby extending the scope of support from the trial level to the experiment level.

A trivial approach would be to allow the engineer to simply ask for some specific number of trial executions to be run as a batch. But given the potentially high cost of deploying and executing large-scale trials, the engineer will want to execute exactly the right number of trials, no more and no less. Of course, this is something that can only be determined dynamically, as the trial results come in.

A better approach, therefore, allows the engineer to schedule repeated trials under a *stopping condition* that performs incremental analysis of the results. Using dynamic stopping conditions based on so-called "significance analysis" is a common practice in the simulation community. Several such statistical conditions have been formulated, and their implementations are readily available. They can be naturally incorporated into the context of live experimentation.

Weevil automates the scheduling, deployment, and execution of a minimal set of trials based on engineer-specified stopping conditions. It offers a library of the most common statistical conditions, but also allows the engineer to plug in new ones. Weevil supports two schemes for deploying and executing repeated trials: sequential and concurrent.

The sequential scheme is the only option when the testbed for a trial is fully determined in the trial configuration, which means that the same trial can only be mapped to the same set of testbed elements (i.e., hosts and network links). In this case, Weevil must iterate sequentially through a series of trials, at each step waiting for the completion of the current trial and for the release of the testbed resources. Upon completion of the post-processing phase of each trial, the chosen stopping condition is evaluated. If the metrics of interest collected up to that point are significant within the desired level of confidence, Weevil stops the experiment. Otherwise, it goes on to schedule the next trial. The disadvantage of this scheme is that it may not exploit all the available testbed resources. However, this scheme also has some advantages. In particular, since the experiment configuration is the same for all the trials, there is no need to regenerate the control scripts, nor redeploy some or all of the scripts and system components between trials. This may significantly reduce the overall time to conduct an experiment.

If the configuration of a trial does not require a specific assignment of testbed elements, the trials can be dynamically mapped onto different elements. In this case, if enough qualified elements are available, Weevil can schedule multiple trials concurrently. Such a scheme can make better use of the available testbed resources. However, since each trial may be mapped onto a different set of elements, different control scripts may have to be generated and deployed for each trial. Furthermore, because of the concurrent execution of multiple trials, the experiment may execute more trials than what is strictly necessary. This is because the stopping condition may be satisfied at the end of a given trial while other, concurrent trials are still in progress.

In practice, for both schemes, the engineer needs to specify a maximum and minimum number of trials, along with a data-analysis procedure (possibly void) that implements the stopping condition. The engineer then specifies the experiment configuration as usual. The configuration may be specified with an explicit testbed mapping, as in the previous version of Weevil [18]. In this case, Weevil schedules multiple trials sequentially. With Weevil's new testbed model, the engineer can also specify the testbed mapping using symbolic resources, in which case Weevil is able to schedule multiple trials for concurrent execution.

4. EXPERIENCE

We validated our enhancements through a series of experiments carried out on the Emulab testbed. The experiments examined the performance and fault tolerance of CFS [5], a wide-area, cooperative, replicated storage service built on top of the Chord distributed hash table [16]. What differentiates the experiments is the operational context of the network and hosts. We defined five such contexts, each one characterized by a specific pattern of environmental phenomena. The first exhibits variable link latency so as to emulate a network of mobile hosts. The others exhibit failures of four different kinds: simultaneous, uniformly random, reactive, and malicious.

Due to space limitations, we are unable to present the first two failure experiments, simultaneous and uniformly random. The first of the two is important because it reproduces in the large the results of a small-scale experiment described by the designers of CFS [5]. The second is important because it extends through experimentation the results of an early simulation exercise conducted by the developers on the Chord network [16]. The interested reader can find details of those two experiments presented elsewhere [17].

The mobility experiment, and the reactive and malicious failure experiments, demonstrate the benefits of increased expressiveness. The mobility experiment also demonstrates the utility of automated multi-trial stopping conditions under both the sequential and concurrent schemes. The malicious failure experiment demonstrates the composability of application and environment workloads. All the experiments demonstrate the way in which our methods support repeatability through control of the operational context.

4.1 Mobility

In this first series of experiments we evaluated CFS in an emulated network of mobile hosts. The environment is configured to appear as though it were a cellular network using GPRS [1], where the movement of hosts conforms to the "random waypoint" mobility model within a single cell [8]. The experiments use a 16-node CFS network with 320 files already resident. Each CFS node has an application actor constantly issuing 50 retrieval requests for uniformly random file keys every 16 seconds. The experimental goal was to evaluate the performance of CFS in transient cellular networks that involve fast host movements, and to compare that to stable networks with slower movements.

In each experiment, link latencies were dynamically modified to simulate the host movements. These modifications were driven by an environment workload that was generated with the help of a mobility trip generation tool [12]. The tool generates trips (the initial location and movements of each mobile host) based on the user-provided constraints

of the moving space, speed, and pause time. Using the trace of the generated trips, we calculated the transient distance between each mobile host and the base station (set to be the center of the rectangular moving space) every five seconds. The latency between each host and the base station, i.e., the latency of the corresponding Emulab LAN link, was then adjusted accordingly using a logarithmic drop of connection speed as a function of the distance. In order to apply the environment workload, we associated an environment actor program with each link. All the actor programs were then deployed and executed on the main Emulab controller host *users.emulab.net*, which is where all network properties can be controlled in Emulab. In this case, an environment actor program consists of a simple iteration that, for each workload line, invokes `delay_config` with the appropriate parameters to modify the latency of its associated link. Using this basic configuration, we ran multiple experiments, each with a different environment workload capturing a different trip and mobility pattern.

Each experiment consisted of three to ten repeated trials, depending on the consistency of their execution results. We used 32 Emulab hosts, and for each trial the 16 CFS nodes were mapped onto 16 of the 32 hosts. As a result of each trial, we measured the range of retrieval response times, recording their 5th, 50th, and 95th percentile. We set the dynamic stopping condition to a 0.1 standard deviation of each of these values, with a minimum of two trials. Since Emulab is a controlled environment, the trial execution results turned out to be very consistent, requiring only three trials when we used the sequential trial-scheduling scheme and four trials when we used the concurrent trial-scheduling scheme. Due to these small number of trials, we observed an advantage of the concurrent scheduling over the sequential scheduling of only approximately 10% in total start-to-finish experiment setup and run time, which includes script generation, deployment, and execution. We hypothesize that the difference would be much more significant in the presence of more trials and/or larger testbeds. To further validate Weevil's trial management, we manually introduced variability in the link latencies. In this case, more trials were automatically scheduled to finally meet the stopping condition.

Figure 6 shows the 5th, 50th, and 95th percentile of the retrieval response times measured in the experiments. The data points for each type of mobile network were observed in five experiments involving 800 block lookups under different environment workloads. The figure shows a surprising result: transient networks exhibit slightly better response times than stable networks. This result contradicts our initial intuition that more and faster movements would affect the CFS network by increasing the response times.

This intuition prompted us to study the scenario further. Examining the environment workloads, we realized that this surprising result is consistent with the overall distributions of transient locations, which are closer to the base station in transient network scenarios. In fact, we found that this is a general property of random waypoint mobility when the base station is in the center of a closed moving space. This property becomes intuitive if one considers the distribution of the points of a random segment in the closed movement domain. Such segments, which are the basis of the random waypoint model, tend to pass closer to the center rather than the periphery of the domain. Therefore, the more segments are added to a trip, the longer the time the host spends closer

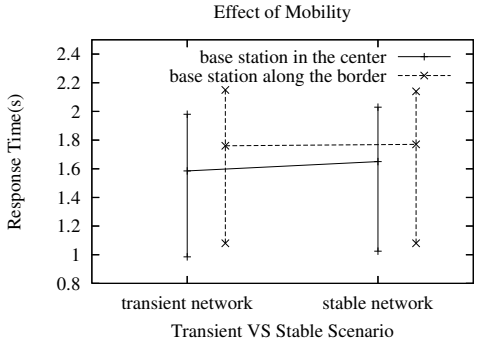

Figure 6: Retrieval Response Time of the CFS Network under Transient Networks and Stable Networks

to the (central) base station. (An analytical description of this property is beyond the scope of this paper.)

To verify this analysis, we regenerated a set of environment workloads by relocating the base station on the border of the rectangular space, with all the other configuration parameters unchanged. The results of these experiments are shown in Figure 6 with dashed lines. When the base station is located along the border, CFS exhibits comparable response times for the two types of mobility scenarios, which is consistent with our analysis. Also, in all experiments, we observed no errors. We conclude that the performance of CFS does not degrade in a mobile environment where hosts move within a certain range, and in fact, depending on the location of the base station, frequent movements can even have a positive effect on response time.

Although some of our conclusions may be valuable to the designers of CFS, the point of these experiments is not to evaluate CFS or to verify what is probably a well-known property of the random waypoint model. Rather, the experiments show that: (1) by constructing environment workloads, we can easily support repeatable experimentation over a wide range of environmental conditions, including ones that might be very different from the basic nature of the testbed; (2) the experiment-level management helped us control the error bar brought by unwanted interruptions; and (3) the automation provided by Weevil facilitates the exploration of different, but related, scenarios.

It is interesting to note that the developers of Emulab have made available a robotic version of their testbed to support experiments in truly mobile environments [9]. This robotic version currently employs a small number of remotely controlled vehicles. Clearly such an approach, while innovative, does not scale in the way our approach can, nor can it account for movements not directly supported by the vehicles, such as high speeds and/or three-dimensional vectors.

4.2 Robustness

Robustness to failures is one of the fundamental properties that any designer of a distributed system hopes to achieve. But it is also a property that is notoriously difficult to establish because of the difficulty in controlling the operational environment. The next series of experiments shows the utility of Weevil in conducting the most common kind of fault-injection experiments, ones in which failures are assumed to be fail stop.

Reactive Fault Injection. Hardware and software faults are frequently caused by incorrect operation or unanticipated heavy load on the system. This series of experiments represents a scenario in which system components with a heavier load have a higher probability to fail. Our experimental goal is to verify that the failure of the heavier-loaded CFS nodes has a worse effect on system performance than the failure of randomly selected CFS nodes.

We conducted the experiments using a 100-node CFS network deployed over a 50-host Emulab testbed, with 2000 files already resident. Each CFS node has an application actor constantly issuing 50 retrieval requests for uniformly random file keys. Since we could not determine which CFS node should fail before trial execution, we associated an environment actor with every CFS node. Each actor's portion of the environment workload is a list of conditional "JOIN" and "FAIL" actions issued at constant intervals. Whether they would be issued or not depends on the two variables `rpcrate` and `liveornot` of the associated CFS node. A scheduled "FAIL" action is effective when the node's remote procedure call (RPC) communication rate `rpcrate` is higher than a specified threshold. A "JOIN" action is executed if the node is in the down status, with `liveornot` equal to 0. In the environment workload, a conditional "FAIL" action appears every 10 seconds and a "JOIN" action every 120 seconds. For example, the following snippet for actor A0 (represented during workload generation as the simulation process U0) is extracted from the generated environment workload when we assign the threshold value 30:

```
event(10,U0,FAIL(),rpcrate -gt 30,rpcrate)
event(20,U0,FAIL(),rpcrate -gt 30,rpcrate)
...
event(120,U0,JOIN(),liveornot -eq 0,liveornot)
...
```

The actor program implements the translation from the conditional actions to the real requests.

We conducted a set of experiments with RPC rate threshold values of 13, 14, 15, and 16. The different threshold values cause different node failure rates over time. For each threshold value, we performed five trials. Each trial runs for 5000 seconds. We separated the log file of each trial into five segments, each of which is 1000 seconds long. For each segment, we calculated the median of the retrieval failure rate caused by the network inconsistency versus the node failure rate over time, and we plotted the results in Figure 7, comparing them with those of the uniformly random-failure experiments [17]. As conjectured, the reactive fault injection scenario exhibits higher retrieval failure rates than the random-failure scenario.

This series of experiments is important because it demonstrates an environment model that is dependent on the dynamic state of the subject system. This kind of behavior could only be modeled using Weevil's enhanced workload language. Such an expressiveness is implemented through a workload containing conditional workload lines, through the actor programs with the ability to translate the workload lines, and the ability to probe the dynamic execution states during the trial run.

Malicious Fault Injection. Through this series of experiments, we demonstrate the composability of workloads. We define scenarios in which failures are maliciously injected into CFS nodes that play crucial roles within an application

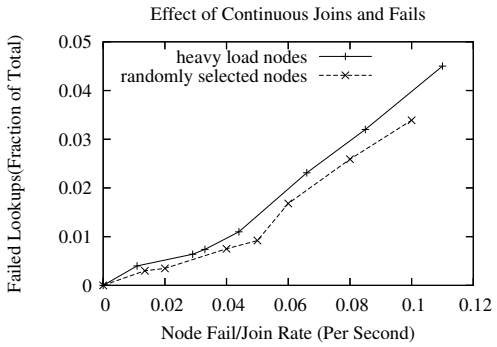

Figure 7: Failing Lookups as a Function of the Rate (over Time) at which Nodes Fail and Join in Uniformly Random (randomly selected nodes) and Reactive Fault Injection Experiments (heavy load nodes)

workload. To explain this model in detail, we must briefly review the internal lookup mechanism used by CFS.

In the CFS implementation, each CFS node maintains a "finger table" to speed up the lookup of a key. All the CFS nodes make up a logical ring used for the Chord routing algorithm. Each CFS node has its logical successor in the ring stored as the first entry in its finger table. Upon receiving a retrieval request, a CFS node always tries to first contact its successor. If the successor is unreachable, it tries to contact the next entry in its finger table. In this scenario, the environment actors are designed to exploit this behavior. Instead of randomly choosing which and when nodes should fail, each environment actor always chooses to shut down the successor of the node that will receive the next incoming retrieval request. In other words, the failures are intended to represent a worst-case scenario or perhaps a denial-of-service strategy of a malicious attacker. We refer to the application requests that are subjected to this attack as "attacked requests". Based on the CFS routing algorithm, we predict that attacked requests should have longer response times, but should not fail.

We reused the subject CFS network, the Emulab testbed configuration, and the application workload used in the reactive fault injection experiments. We also reused the environment workload from the uniformly random fault injection experiments. The application and the environment workloads are originally independent. In this malicious scenario, we adjusted the environment workload to always bring down the successor to the node to which a retrieval request is going to be routed. Figure 8 exemplifies the inputs of the two types of workloads and the adjusted environment workload.

The original environment workload schedules the "FAIL" actions for the associated system components at times 242 and 439. Through the workload weaving, we wish to adjust the workload to attack ("FAIL") the successor to which a future request, issued 5 seconds later, will be routed. The 5-second delay is a necessary parameter of the fault-injection mechanism, as a CFS node needs several (less than 5) seconds to shut down. Thus, the "FAIL" action at time 242 should attack the retrieval request `event(247,U48,GET(f1900))` in the application workload. According to the trial configuration (Figure 8c), the

```
event(242,U42,GET(f1799))
...
event(247,U48,GET(f1900))
...
event(444,U12,GET(f1561))
```
(a) application workload

```
event(242,I0,FAIL())
event(439,I1,FAIL())
...
```
(b) environment workload

```
'U'i <-> 'N'i
'I'i <-> 'N'i
i=0..99
```
(c) trial configuration

```
event(242,I48,FAIL(succ),,succ)
event(439,I12,FAIL(succ),,succ)
...
```
(d) resulting environment workload

Figure 8: Workload Weaving. The workload in (d) is derived from (a) and (b) woven according to (c).

client U48 sends all requests to the system component N48. So the environment workload action at 242 is adjusted to be sent from the actor I48, who is also mapped to N48, to attack the successor to N48. The workload line becomes `event(242,I48,FAIL(succ),,succ)`, where `succ` is the parameter to be assigned during trial execution.

The adjusted environment workload is applied through 100 environment actors, each of which is associated with a CFS node. Each environment actor is in charge of shutting down its associated CFS node in response to an appropriate signal by another environment actor. Specifically, the environment actor associated with the application actor that sends out the attacked retrieval request sends a shutdown signal to its successor. An actor can find the identity of its successor using the Chord client program `getsucc`. Thus, the determination of whether or not an environment actor should bring down or start up a CFS node is dependent on the trial-execution-time response to the `getsucc` program and the inter-actor communication message.

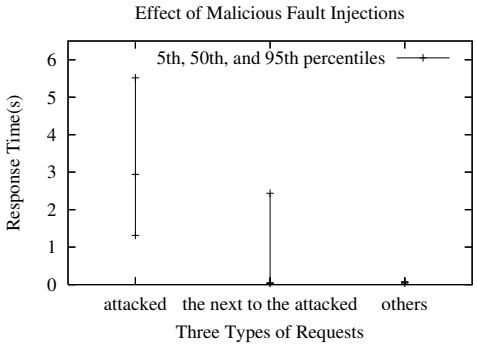

Figure 9: CFS Retrieval Response Time under Malicious Attack

As expected, we observed no failure for the requests because of the loss of successors. Some requests failed because of the inconsistency in the CFS network's routing tables during their fixing process. We divided all the retrieval requests into three types: the attacked requests, the next requests sent by the actors whose last requests were attacked, and all others. The purpose of dividing them in this way is to study the effect of the successor loss and to study if the effect lasts. The results are shown in Figure 9, which gives the 5th, 50th, and 95th percentile of the retrieval response times for the three types of requests. The retrieval response times of the attacked requests were longer than the non-attacked ones because of the unavailability of the successors.

This series of experiments validates the utility of the workload composition mechanism in reusing the workloads for simple experiment scenarios to obtain those for a broader class of experiment scenarios. The simple format of workloads makes their composability tractable and straightforward, certainly more so than if the workloads were full-blown programs.

5. RELATED WORK

The basic Emulab management system automatically allocates hosts, implements network configurations, and executes scheduled actions. A recent addition to Emulab is the Experimentation Workbench [6]. Its contribution is mainly in the *recording* of relevant environment parameters of each trial to enable replayable execution, which is a feature complementary to Weevil's ability to *control* the environment. It also supports automated sequential execution of multiple related trials. Overall, however, its features are tied to the specifics of the Emulab environment, and so is not a general-purpose tool. A further limitation is that it does not provide any direct support for workload generation. Finally, it is limited to static behaviors, since all actions in the management system must be prescheduled during setup.

DART [4] is an automated regression testing framework built on Emulab. It provides a set of primitives for writing tests for distributed systems and a run-time mechanism to execute the tests in a fast and efficient manner. ACME [11] is a framework for automated robustness evaluation of distributed services. It is targeted at both emulated network testbeds, such as Emulab, and real wide-area testbeds, such as PlanetLab. ACME considers only component failures and high network usage in failure scenarios. The faults are controlled, monitored, and injected through per-node sensors and actuators. Reactive scenarios are provided in ACME through the sensors. However, neither DART nor ACME consider the behaviors of the clients of a distributed system. Moreover, as in the Experimentation Workbench, they require that all actions be prescheduled.

Plush [2] and PlMan[5] are execution management systems for PlanetLab. Users describe what amounts to a trial in an XML document in Plush, or as scripts in PlMan. The descriptions are used to automatically prepare PlanetLab hosts, deploy the subject system, execute commands, monitor progress, and handle any clean up. However, these tools provide no support for the other activities of experimentation, such as workload generation, nor do they provide direct support for modeling and executing distributed client behavior. Of course, it would be conceivable to retarget Weevil's model-based script generation facility to make use of the Plush or PlMan trial execution mechanisms.

JMeter[6] is a performance testing tool designed for the Apache Web server. Among its features is the ability to support conditional (reactive) workloads. The workloads are formed from a set of specialized modules, called controllers, that embody the logic of a Web client. Weevil implements a similar capability, but through a mechanism that is deliberately more restricted. These restrictions allow the engineer to structure the results of the workload generator into a variety of different *distributed* workloads, something for which the JMeter logic-controller approach is not appropriate.

[5]http://www.cs.washington.edu/research/networking/cplane
[6]http://jakarta.apache.org/jmeter/

StarBED,[7] another emulated network testbed, uses VLAN configurable switches to enable the configuration of network properties. Its developers created a tool called SpringOS [10] to automate its usage. SpringOS is targeted at experiments with client/server systems, where it can model client behaviors that depend on server responses. However, SpringOS is specific to StarBED and, moreover, does not support dynamic environment variations.

The most common environment variation scenarios seem to be failure scenarios. Since failures may occur due to correlated distributed faults, many fault injection techniques and tools have been developed to help engineers create realistic failure scenarios [7]. One such system is Loki [3]. Loki is deployed to form a framework for maintaining a partial view of the global state, and for injecting faults based on that partial view. The partial view is derived using state machines associated with the communicating Loki nodes.

6. CONCLUSION

The work described in this paper is intended to facilitate the evaluation and tuning of network protocols, large component-based systems, and distributed systems in general through repeatable experimentation in realistic environments. We have presented four significant enhancements to our original methods that allow an engineer to explore richer scenarios and to obtain results with greater confidence. Full control of the environment by means of environment workloads is the basis for repeatability. Expressiveness of the workloads and the weaving of application and environment workloads are the key ideas to obtain rich scenarios. Automated multi-trial experiments are essential to obtain statistically valid measurements. We demonstrated the validity of these ideas through a series of challenging experiments carried out on a popular state-of-the-art testbed.

In the future we will continue to work to improve the way experiments are conducted on large-scale software systems. The natural evolution of this research is to expand our focus from a single experiment to a whole series of experiments. In particular, we plan to study how to better characterize system behaviors by automating the exploration of the parameter space. Our intuition is to view this as an identification and optimization problem.

7. REFERENCES

[1] 3rd Generation Partnership Project (3GPP). *General Packet Radio Service (GPRS); Service description; Stage 2*, June 2006. TS 23.060, V. 7.1.0.

[2] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. Planetlab application management using Plush. *ACM Operating Systems Review*, 40(1):33–40, Jan. 2006.

[3] R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders. Loki: A state-driven fault injector for distributed systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 237–242, New York, NY, June 2000.

[4] B. N. Chun. DART: Distributed automated regression testing for large-scale network applications. In *Proceedings of the 8th International Conference on Principles of Distributed Systems*, pages 20–36, Grenoble, France, Dec. 2004.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 202–215, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[6] E. Eide, L. Stoller, and J. Lepreau. An experimentation workbench for replayable networking research. In *Proceedings of 4th USENIX Symposium on Networked Systems Design and Implementation*, pages 215–228, Cambridge, MA, Apr. 2007.

[7] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[8] E. Hyytia and J. Virtamo. Random waypoint mobility model in cellular networks, 2005.

[9] D. Johnson, T. Stack, R. Fish, D. M. Flickinger, L. Stoller, R. Ricci, and J. Lepreau. Mobile Emulab: A robotic wireless and sensor network testbed. In *Proceedings of the 25th Conference on Computer Communications*, Barcelona, Catalunya, Spain, Apr. 2006.

[10] T. Miyachi, K. ichi Chinen, and Y. Shinoda. Automatic configuration and execution of Internet experiments on an actual node-based testbed. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, pages 274–282, Trento, Italy, Feb. 2005.

[11] D. Oppenheimer, V. Vatkovskiy, and D. A. Patterson. Towards a framework for automated robustness evaluation of distributed services. In *Proceedings of the 2nd Bertinoto Workshop on Future Directions in Distributed Computing: Survivability: Obstacles and Solutions*, Bertinoro, Italy, June 2004.

[12] S. PalChaudhuri, J.-Y. L. Boudec, and M. Vojnovic. Perfect simulations for random trip mobility models. In *Proceedings of the 38th Annual Simulation Symposium*, San Diego, CA, Apr. 2005.

[13] 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.

[14] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. Analysis of caching and replication strategies for web applications. *IEEE Internet Computing*, 11(1):60–66, Jan. 2007.

[15] N. Spring, L. Peterson, A. Bavier, and V. Pai. Using PlanetLab for network research: Myths, realities, and best practices. *SIGOPS Operating Systems Review*, 40(1):17–24, Jan. 2006.

[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*, pages 149–160, San Diego, CA, Aug. 2001.

[17] Y. Wang. *Automating Experimentation with Distributed Systems Using Generative Techniques*. PhD thesis, University of Colorado at Boulder, Boulder, CO, Aug. 2006.

[18] Y. Wang, M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Automating experimentation on distributed testbeds. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 164–173, Long Beach, CA, Nov. 2005.

[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.

[7] http://www.starbed.org/