# Automating Experimentation with Distributed Systems Using Generative Techniques

by

**Yanyan Wang**

B.S., Peking University, Beijing, China, 2001

M.S., University of Colorado at Boulder, 2003

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2006

This thesis entitled:
Automating Experimentation with Distributed Systems Using Generative Techniques
written by Yanyan Wang
has been approved for the Department of Computer Science

Prof. Antonio Carzaniga

Prof. Alexander L. Wolf

Date ________________

Wang, Yanyan (Ph.D., Computer Science)

Automating Experimentation with Distributed Systems Using Generative Techniques

Thesis directed by Prof. Antonio Carzaniga and Prof. Alexander L. Wolf

Engineering distributed systems is a challenging activity. This is partly due to their intrinsic complexity, and partly due to the practical obstacles that developers face when evaluating and adjusting their design and implementation decisions. This thesis addresses the latter aspect by providing a framework to automate experiments. The experiment automation framework is designed in a generic and programmable way to be used with different types of distributed systems for wide-ranging experimental goals. It covers three key steps for each experiment: (1) workload generation, (2) experiment deployment and execution, and (3) post-processing. We designed an approach to workload generation, the simulation-based approach, in which the stimuli of the subject system are modeled by simulating its user behaviors and its execution environment variations. The execution trace of the simulation programs constructs a workload. We automate the next two steps with a model-based generative approach. It is founded on workloads and a suite of configuration models that characterize the distributed system under experimentation, the testbed on which the experiment is to be carried out, and their mappings. The models are used by generative techniques to automate construction of a control system for deploying, executing, and post-processing the specific experiment. We have validated our approaches by performing experiments with a variety of distributed systems on different testbeds to achieve wide-ranging experimental goals. Our experience shows that this framework can be readily applied to different kinds of distributed system architectures and distributed testbeds, and that using it for meaningful experimentation, especially in large-scale network environments, is advantageous.

**Dedication**

In memory of my dear grandma, who is always in my heart.

## Acknowledgements

First of all, I would like to express my deepest thankfulness to my advisors, Antonio Carzaniga and Alexander L. Wolf, who guided me through the whole process. I could not imagine achieving so much when I first came to them four years ago. Antonio could always quickly grasp my thinking and propose interesting ideas even if I myself was still unclear and hesitating. Alex was always there to guide my work into a correct direction from a researcher's point of view. They always motivated me in an encouraging way, not only on my research, but also on my personal improvement in communicating with other people.

I would like to thank all of my fellow members of SERL for their continuous comments on my work and those sweet memories we share. Especially, I thank Dennis Heimbigner for the beneficial SERL meetings and his valuable feedback about my work. Thanks to Naveed Arshad for sharing his experience in technical writing. Thanks to Nathan D. Ryan for being the main force in moving the SERL lab. Thanks to Laura Vidal for everything fun and everything complicated. A special thanks goes to Matthew J. Rutherford and John Giacomoni, who I bothered the most. Thanks to Matt for the valuable input and pleasant cooperation, for the great PlanetLab, for Gentoo Linux, and for his warmness in helping me get out of many troubles. Thanks to John for enlightening me on Unix and for the wonderful parties and the delicious homemade food. I feel so lucky to have them as my companions all the way.

I also got directions and care from many other faculty members. I would like to thank John Bennett for his skepticism, which promoted me to be more serious and motivated. Thanks to Shivakant Mishra for his willingness to help in person. Thanks to C. Edward Chow for his encouragement and the long distance drives. Thanks to Skip Ellis and Leysia Palen for being like good friends and caring about my every little accomplishment.

Last but not least, my highly appreciation to my parents and my husband. Thanks to my parents for letting their only daughter to go far away to pursue her dream. They give all of their love to me with only the wish for me to be happy and healthy. I can still feel their love overseas. Thanks to my husband Yuxiang for giving me a home in US. He is always there to unconditionally support me and share my happiness as well as my tears. Thanks to all of the others the stories of whom are too long to be included in this limited space.

# Contents

# Tables

## Figures

# Chapter 1

# Introduction

With the increasing need of Internet-scale services, development of their underlying scalable and reliable distributed systems has been emphasized in recent years.[1] A distributed system usually consists of a network of components, executing independent and possibly heterogeneous tasks. Various architectures are used for distributed systems. In the traditional client/server architecture, each component (server) independently offers service to multiple, distributed clients. In more complicated architectures of distributed systems, components cooperate to collectively realize a coherent service serving many clients through a large number of distributed access points. We call distributed systems with such architectures **highly distributed systems**. 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. Their architectures are much more complicated than the simple client/server architecture.

Because of the complexity of distributed systems, software engineers face challenges all along their development process. Early on in the process, they face difficulites in validating the intrinsic correctness and practicability of their large, distributed architecture and protocol design. By contrast, the challenges later on in the process focus more on the low-level design parameters and implementation details. For example, to

[1] http://dsonline.computer.org/portal/site/dsonline/index.jsp

improve system performance for most common cases, the engineers need to identify and optimize hot spots in the design details. To test the correctness of their system implementation, they need to analyze execution results for sets of test cases.

For the challenges early on in the development process, formal methods and simulations can offer valuable guidance to the engineers. The formal method verifies distributed designs using arguments based on mathematical models. Example technologies of the formal method are formal specification, model checking, and performance modeling. Another more flexible and thus more popular method, simulation, is based on operational models of a distributed system design, the system's clients, and operating environments. Although both formal methods and simulations can provide valuable information about high-level design alternatives, the abstract models that they are based on often fail to capture important details of actual system behavior. Thus, they are not the proper solutions to the difficulties typical of the later development stages, in which details and accuracy become essential. Instead, software engineers must conduct systematic, repeated experimentation with executable prototypes of a distributed system in realistic execution environments to yield more accurate results.

An experiment in this context is "a rigorous, controlled investigation of an activity, where key factors are identified and manipulated to document their effects on the outcome." [24] Specifically, each experiment consists of a number of closely related **trials** of a particular distributed system with different values for the key fractors. For instance, an experiment aimed at evaluating the performance of a web proxy might consist of a number of trials whose key factors are the parameters controlling its caching policy. As illustrated in Figure 1.1, there are several steps in carrying out experimentation [24]:

(1) **Conception**: to analyze the features of the system under experimentation (SUE) and decide the goals of the experiment;

(2) **Design**: to state an idea and generate a design to test the idea. The experi-

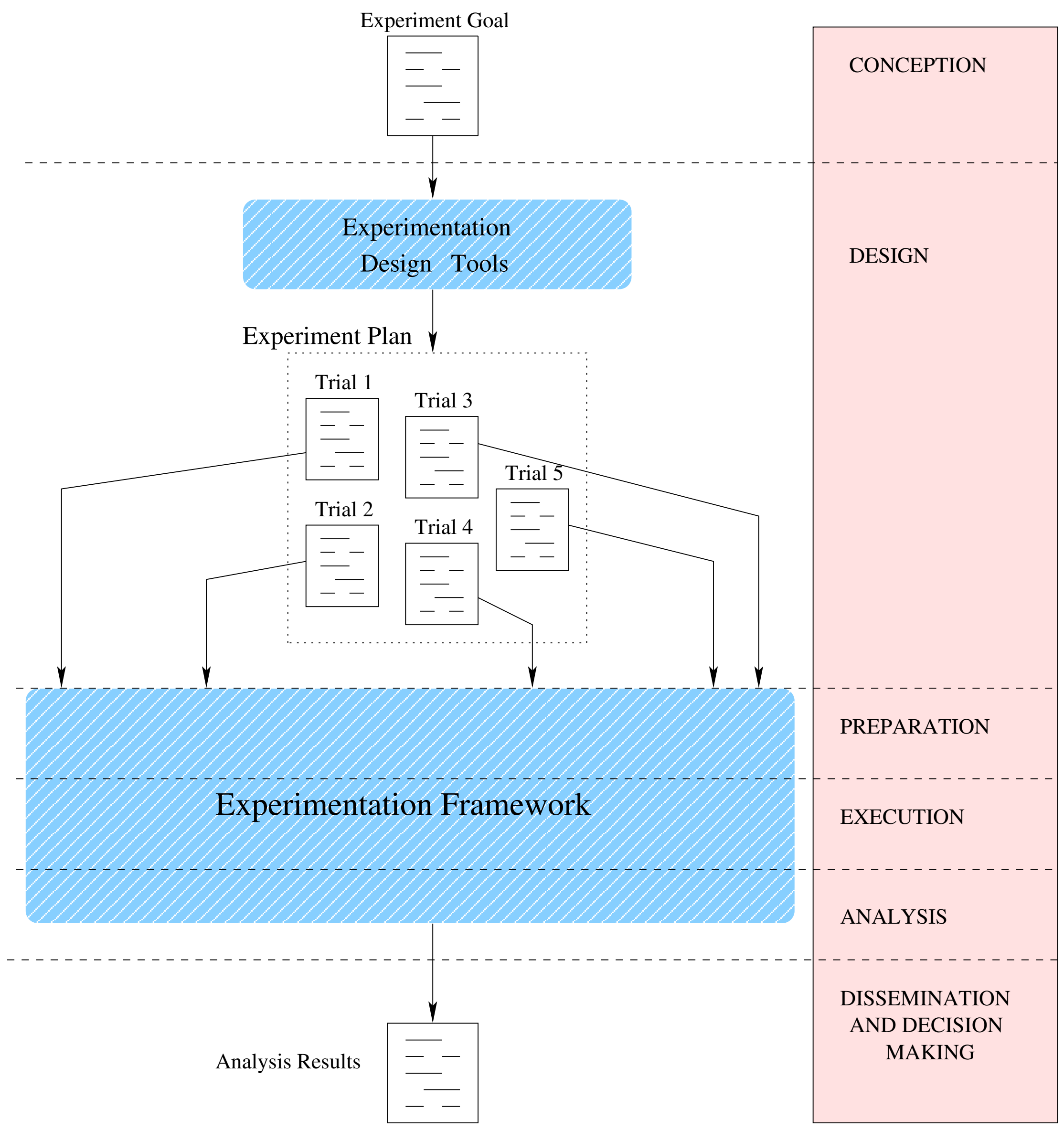


Figure 1.1: Six-Step Experimentation Process to Evaluate a Software System. The three steps, preparation, execution, and analysis, are repeated for each trial in an experiment.

mental design written into an experimental plan describes how experiment trials will be organized;

(3) **Preparation**: to ready the SUE and the environment for each trial of the experiment;

(4) **Execution**: to execute each trial of the experiment following the steps laid out in the experimental plan;

(5) **Analysis**: to review the experiment and analyze data, including analysis of each individual trial and analysis of the global experiment;

(6) **Dissemination and Decision Making**: to document analysis results and make decisions.

Many real issues require the ability to conduct this experimentation process for distributed systems as quickly as possible. The growth of software development tools has helped software engineers dramatically improve their productivity in software development. They can now deliver software faster. But this also increases the pressure on the software evaluation activity. They need to dramatically improve their efficiency in conducting experiments. Furthermore, rapid and unpredictable change of network status will threaten to make the experimentation results obsolete before the experiment is conducted. For example, some assumptions about traffic mix, topology or protocols of the network might only be valid for less than a few months [39].

Unfortunately, most of the existing experiment tools [33, 55] targeted to promote the experimentation process are oriented toward much simpler scenarios, such as client/server systems. The others [13, 49] are inflexible and hard to use. They still require software engineers to provide hand-written experiment scripts. For an experiment with a distributed system, the scripts need to solve the complexity brought by the scale, heterogeneity, and instability of the system as well as the execution environments. They are hard to manage and experiment-specific.

The vision of this thesis is to support systematic, repeated experimentation with distributed systems, especially highly distributed systems, in realistic network environments by creating a consistent, unified, reliable experimentation framework. For any experiment with a distributed system, this framework helps software engineers quickly setup each trial control system that is customized to the specific trial, and easily manage the execution and analysis of the trial. Through it, software engineers can achieve efficient cost savings in conducting experiments by automating their experiment trials. As shown in figure 1.1, this framework assumes the availability of an experimental plan and is targeted at promoting the experiment preparation, execution and analysis steps for the experiment trials.

The prototype framework named **Weevil** [2] serves as a foundation for research in fulfilling the vision. Weevil implements the framework to automate experiment trials of distributed systems performed in realistic network environments. It supports trial preparation by generating trial workloads and control scripts and deploying them onto a network together with the SUE. It supports trial execution with the generated control scripts that automatically control and coordinate the execution of the deployed distributed resources. And last, it supports trial analysis by collecting and analyzing timestamped experiment log files.

## 1.1 Research Motivations

Research into automating experimentation with distributed systems is motivated by a variety of factors. We discuss the most important factors in the following sections.

[2] http://serl.cs.colorado.edu/ ywang/weevil/index.html

### 1.1.1 Challenges in Experimentation with Distributed Systems

Distributed systems pose significant challenges in terms of experimentation. These challenges make it difficult to efficiently conduct the experiments manually. We discuss them in the following three aspects.

**Scale**

A distributed system is distributed in scale and function. In many real applications, a large number of system components are deployed and executed in a distributed manner in wide-area network environments. As a result, the clients of a distributed system are also highly distributed geographically. It is a big challenge to coordinate execution and communication of these remote resources in the experiments. The distributed property of a distributed system also poses a big challenge in handling distributed components' execution status and experiment results, which are recorded during experiments.

**Heterogeneity**

Real network environments are heterogenous. They can be composed of complex topologies with channels of very different capacities changing from 56K to 100M, and computers with different operating systems, time zones, software environments, etc. A distributed system is usually deployed in such environments. To obtain realistic measurements, these issues of heterogeneity should all be considered in the experiments.

Because of its distributed architecture, the clients of a distributed system are hard to control. They show heterogeneous behavior in communicating with the system. How to generate realistic workloads considering such heterogeneous client behavior is still an unsolved research question.

In addition to handling heterogenous behavior of distributed systems, our research motivation also comes from the heterogeneity of distributed system designs. There have been many distributed systems exemplified early on in this chapter. Each of them takes different architecture and protocol designs. We want to demonstrate the feasibility of providing a consistent, unified framework so that software engineers can quickly and easily experiment with various distributed systems.

**Instability**

Real network traffic is constantly changing over time both in volume and statistical properties [39]. Unexpected faults and changes may happen. These accidents of the network will definitely affect the experiments conducted on it. How to handle accidents and how to evaluate robustness of the SUE by intentionally injecting accidents are both interesting research questions.

### 1.1.2 Popularity of Distributed Testbeds

With the increasing need from distributed system engineers and researchers, a number of overlay networks and emulated networks, such as PlanetLab [50] and Emulab [61], have been built up and actively maintained to serve as the testbeds for the engineers and the researchers to exercise distributed systems under realistic network environments. All of these testbeds have a large number of active users. For example,

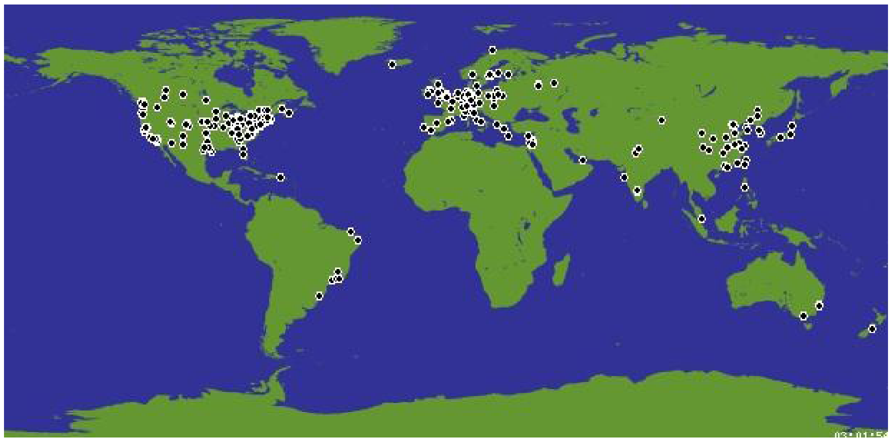


Figure 1.2: PlanetLab Overview. PlanetLab is an overlay network consisting of over 691 nodes located at over 335 sites around the world as of July 12, 2006. Its stated purpose is to provide an "open platform for developing, deploying, and accessing planetary-scale services." See http://www.planet-lab.org/.

PlanetLab [50] as shown in Figure 1.2 is a popular overlay network distributed widely around the world. Its stated purpose is to provide an "open platform for developing, deploying, and accessing planetary-scale services." It provides a decent Internet environment for studying distributed systems. The PlanetLab maintainers and users have contributed a number of tools and services to streamline its usage. But these tools and services are more focused on distributed service deployment and maintenance than on systematic, repeated experimentation, which is our vision of this thesis. The distributed software deployment is only one of the many activities of an experiment trial, so the available supports of PlanetLab are inadequate to easily manage experiment trials running on it.

Another actively used testbed for distributed systems is Emulab [61]. Emulab keeps about 350 machines as of July 12, 2006. Most of the time, more than 300 of them are in use to run experiments. Emulab also has linkage to the PlanetLab nodes. Emulab provides an integrated emulation and simulation environment where the user can configure the network cluster made up of those testbed nodes to have the desired network profile. Although Emulab is not really widely distributed around the world, it provides a more controllable network environment for software engineers than PlanetLab. Thus, software engineers conduct many experiments studying the distributed protocols and algorithms under different network properties. But again, its associated Emulab management system is not targeted at automating experiments with distributed systems and does not solve all the challenges stated in Section 1.1.1.

### 1.1.3 Limitations of Available Tools

Some tools (e.g., DECALS [33], RiOT [55], and TestZilla[3] ) exist to manage the testing of distributed systems. However, they are essentially control and logging systems for executing user-provided test scripts in parallel. To use them, software engineers need

[3] http://www.cs.cornell.edu/vogels/TestZilla/default.htm

to provide workload and to construct large quantities of heterogeneous experiment trial scripts. This makes existing tools hard to use for systematic experimentation with distributed systems. Additionally, none of the documentation available for these tools report evaluation of their operation on large-scale experiments with highly distributed systems.

However, two other testing tools, DART [13] and ACME [49], are targeted at large-scale testbeds like PlanetLab. They do provide scalable framework applicable for experiments with highly distributed systems. But they do not provide mechanisms for software engineers to quickly get all the experiment scripts ready for repeated, systematic experimentation. Neither do they provide flexible mechanisms for constructing complicated experimental scenarios.

Some other testing tools do provide flexible configuration mechanisms for constructing different experiment scripts automatically, but their configuration models are fairly narrow in scope. They are targeted at specific architectures rather than generally applicable to different types of distributed systems. Two examples are the JXTA Distributed Framework[4] and CCMPerf [38]. They are created for JXTA applications[5] and CORBA (Common Object Request Broker Architecture) applications, respectively.

## 1.2 Our Experimentation Framework

As shown in Figure 1.1, our experimentation framework covers the preparation, execution and analysis steps for each experiment trial. As a part of the preparation step, workload generation is the activity to prepare realistic workload inputs to provide stimuli to the system execution. This is an individual research topic for distributed systems [7, 66, 6, 31, 20]. In this thesis, we separate this activity out of the prepartion step and design a framework covering a three-step process of **workload generation**,

[4] http://jdf.jxta.org/
[5] http://www.jxta.org/

**trial deployment and execution**, and **trial post-processing**. Either of these three steps has been under study as an individual research topic by software researchers. Many techniques and approaches have been applied as described in Chapter 2.

In this thesis, we learn from the related work and create a consistent, unified framework to automate the experimentation activity in engineering distributed systems. We have two hypotheses in building up this framework. First, we proposed a **simulation-based workload generation approach** to model workload scenarios using discrete-event simulation. The workload can then be generated by running the simulation program, with the execution trace serving as the synthetic workload. Second, we predicted that any experiments with distribted systems can be abstracted in a suite of models. By configuring these higher-level models, an experiment trial can be specified. The low-level, customized trial scripts automating the whole trial process can be automatically generated using generative techniques. We call this approach **model-based generative approach**. Based on these two hypotheses, our experimentation framework can facilitate experimentation activity for distributed systems by providing engineers with a flexible, configurable, automated and, thus, repeatable process for evaluating their distributed systems on distributed testbeds. Further, it is not targeted at a specific system or application model, but rather it is generic and programmable. Figure 1.3 shows the three steps of a trial with the boxes in different colors. As shown

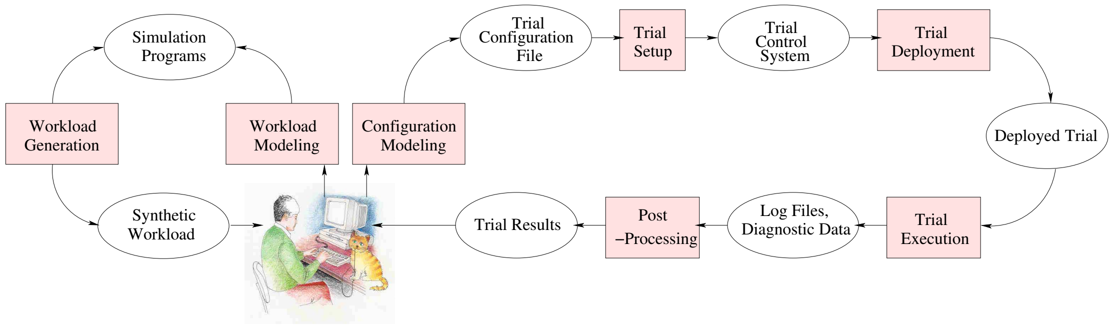


Figure 1.3: Automated Trial Process

in the figure, through our framework, the software engineer only needs to interact with two modeling phases, **workload modeling** and **configuration modeling**. All the other phases to finally return the trial results are automated by our approaches.

A trial in any experiment for studying and evaluating a distributed system is related to four primary concepts: the **SUE**, the **testbed**, the **user actors**, and the **environment actors**. The SUE is the subject of the experiment trial. The testbed is the network environment in which the trial is performed. Both the two types of actors are the actuators of the experiment trial. Each user actor drives the execution of the SUE by injecting system service calls to its system access point. It represents a user using the distributed system. Its workload, **user workload**, is a partially ordered list of system service calls describing the user behavior. By contrast, an environment actor interacts with the trial execution environment with the goal of changing the trial execution environment. Environment actors are used in the experiments when the SUE needs to be studied under the changing network environment (the testbed) or the changing system configuration (the SUE). Its workload, **environment workload** is a partially ordered list of actions changing the SUE or the testbed.

### 1.2.1 Workload Generation

The workload generation generates a user workload for the user actors and optionally an environment workload for the environment actors. These workloads reflect the goal of an experiment. For example, to evaluate the SUE's performance in meeting the service requests, a user workload is generated as a list of service calls from the user actors to the SUE. To evaluate the SUE's robustness under sudden hardware or software failures, in addition to the user workload, an environment workload including a list of fault injections from the environment actors to the testbed is necessary.

We provide an additional, complementary workload generation approach to the currently available workload generation approaches, namely **simulation-based work-**

**load generation approach**, shown as the left circle in Figure 1.3. Its novelty is that it is powered by discrete-event simulation. It can be used to generate both the user workload and the environment workload. Our discussions in this section are focused on the user workload. The environment workload can be generated in the similar way. Using simulation-based workload generation approach, the user workload can be **defined** by simulating each typical type of SUE user behavior. All the individual simulations are assembled into an aggregated simulation program. The workload is **generated** by running the simulation program to produce an execution trace embodying the synthetic workload. The simulation is executed offline before trial deployment and execution. It requires neither the availability of an application nor the results of a statistic analysis, although both can be used in the construction of the simulation. All it needs are models describing how the typical users use the distributed system.

Then, during an experiment trial, the real requests corresponding to the actions in the user workload are issued as service calls to the SUE by actor programs; the same workload can be interpreted differently and reused in different experimental scenarios with the different actor programs. The user behaviors in issuing requests may not be affected by the trial execution status. Then all the actions listed in the user workload are interpreted and injected unconditionally during trial execution. Otherwise, if a user behavior is dependent on the system responses to some previous requests, the workload may contain some conditional actions of which the interpretation needs the trial-execution-time system responses as inputs.

### 1.2.2 Trial Deployment and Execution

Upon readiness of the workloads, in the trial deployment and execution step, they should be applied to the SUE in a network environment. To achieve this, the SUE, the workload, and other trial files (including trial scripts, programs, libraries and auxiliary files) are first deployed across a distributed testbed. Next, the SUE is started and then stimulated by the execution of service calls at the times and locations dictated by the user workload. At the same time, the trial execution environment is modified by the execution of requests at the times and locations dictated by the environment workload. The control of trial deployment and execution is complicated and case-specific.

To achieve complete automation while maintaining its configurability and adaptability, we have taken the **model-driven generative approach** to automate trial deployment and execution as well as post-processing steps. It is based on a suite of models of the desired experiment trial. As shown in the right circle of Figure 1.3, in the first **Configuration Modeling** phase, by configuring these models in a trial configuration file, an engineer can concisely describe an experiment trial of a distributed system. These descriptions are then processed in the **trial setup** phase to generate all the files making up a trial control system adaptively that automate the **trial deployment** and **trial execution** phases, and the **post-processing** step. This approach provides a higher level service for software engineers by requiring much more clean and concise model configurations instead of many complicated trial scripts and programs. For another experiment trial, the engineer only needs to modify several lines of configurations instead of thousands of lines of codes all over the trial scripts and programs. We describe the configuration models in Chapter 5. Then, we introduce in Chapter 6 the automated trial process based on those models.

### 1.2.3 Trial Post-Processing

Although the post-processing step is included in our framework, its main activity distributed data collection and analysis as a whole is another research topic. We chose not to cover it in this thesis work. Instead, we implemented it in a simple way.

After all the actors finish processing their workloads or when the trial reaches the timeout, the trial is terminated. Based on the trial configurations, the post-processing scripts are generated based on the SUE and actor configurations. These scripts are executed against the log files in different locations to produce diagnostic data. Finally, the diagnostic data are sent back to the master machine. The testbed hosts are cleaned up if necessary.

## 1.3 Contributions

This thesis makes a number of contributions in the area of experimentation with distributed systems. The primary contribution is the idea of providing a higher level service than other available work using generative techniques. By applying these techniques, our clean and concise configuration modeling replaces complicated, error-prone script programming. Practical obstacles that hinder experimentation, such as scale, heterogeneity, and instability, are transparent to software engineers and removed by automated script construction. The automation also provides efficient cost savings in conducting experiments. As a by-product, the suite of the trial configuration models is a contribution for defining and understanding an experiment trial of a distributed system. It unifies the trial descriptions of a variety of distributed systems, which makes our trial automation framework generic and programmable, not targeted at a specific system or a specific testbed.

We divide an experiment trial into three relatively independent steps: workload generation, trial deployment and execution, and trial post-processing. This separation streamlines the experimentation activity of distributed systems. Rather than just utilizing the existing technologies for the three steps, we contributed our approaches in the first two. Besides the model-based generative approach we discussed in the last paragraph, we also contributed a more generic, flexible simulation-based workload generation approach to better serve different distributed systems and quickly model emergent scenarios. This approach offers a complementary means to the available approaches. It offers a natural way for software engineers to represent complicated actor behavior at any level of abstraction. Also, it is scalable in the sense that it can seamlessly deal with very complex scenarios, consisting of up to thousands of actors, executing as long as necessary. This ability to scale up is particularly beneficial to distributed systems, which serve a large number of clients and of which the execution environments may be affected by many distributed factors.

The architecture and implementation of the prototypical experiment trial automation framework, **Weevil**, provides additional contributions to the field of experimentation with distributed systems. The prototype explores the effectiveness of the generative approach, specifically, the macro expansion technique, to support experimentation with distributed systems in a unified, consistent manner. The mechanism applied in the prototype has exhibited great scalability, flexibility and reliability in conducting experiments in wide-area network environments. It compensates for the shortage of support for repeated, systematic experimentation on the large-scale, unreliable testbeds such as PlanetLab. And it has provided valuable lessons and insights throughout its implementation.

## 1.4 Evaluation

To validate that the work in this thesis really makes our claimed contributions, we have thoroughly evaluated our implemented prototype, using case studies and meaningful experiments. Specifically, our evaluation is divided into the following five parts.

**Fidelity of the workload models** We are interested in learning if our simulation-based workload generation approach can really support software engineers in modeling different distributed system experimental scenarios. In other words, we are asking the question: is our workload-generation method faithful to specific problems and scenarios?

To answer this question, all the workloads used in our experiments were generated with our implemented simulation-based workload generator. In specific, through case studies, we evaluated the faithfulness of the generated workloads versus the real execution traces. For example, through straightforward programming, we generated workloads that capture the characteristics of the CodeRed-like worm propagation trace. Moreover, through an experiment with a web caching system, we were able to reproduce and broaden the results of a published study on cooperative web caching [62] without incurring the cost and difficulty of collecting additional live trace data.

**Versatility of the trial configuration models** Our second hypothesis assumes that a suite of trial configuration models is able to define and fully describe an experiment trial with a distributed system. To verify that the model is widely applicable, we are asking the question: Is our suite of trial configuration models applicable to a wide variety of distributed systems, testbeds, and experimental scenarios?

To answer this question, we conducted many case studies and experiments in which we:

- modeled two different publish/subscribe communication systems (Siena [10] and Elvin [53]), a mobility service for publish/subscribe clients (MobiKit [8]), two different peer-to-peer file sharing systems (Freenet [15] and Chord [54]), and a composite web-cache system (Squid proxies [27] and Apache web servers [40]);
- modeled three different deployment and execution environments (a local-area testbed, a wide-area testbed on PlanetLab, and an emulated testbed on Emulab);
- modeled four different fault injection scenarios, and conducted robustness testing under these scenarios.

**Scalability of the deployment and execution mechanisms** Our research of the trial automation framework is targeted at highly distributed systems. Thus, an important metric of the framework is its scalability: Is our framework capable of handling large-scale experiments and testbeds?

To answer this question, we performed two experiments on the PlanetLab planetary-scale testbed. Through the first experiment, we validated experimentally the results of an analytical analysis of Freenet [14] that predicted the behavior of a new routing algorithm. Through the second, we provided a more realistic experimental analysis of scalability in Chord than that of a published study [54] by performing an experiment that used a similar number of distributed components deployed over an order of magnitude more machines.

**Utility of the automation features**

The research question we are asking in this part is: Does our model-based generative approach provide effective cost savings in automating the experimentation process? We thus collected six metrics of measuring the model configuration input, of the generated output, and the time cost of some experiments. These metrics confirmed the advantage of the automation feature brought by our model-based generative approach.

**Support of unreliable testbeds**

The last research question is: Is the reliability mechanism that our framework provides able to correctly and efficiently support experiments on unreliable testbeds like PlanetLab? We thus measured the cost to revise and restart an experiment when unexpected failure is introduced to the trial testbed. The cost turns out to be minor.

## 1.5 Organization of this Thesis

In Chapter 2 we give an overview of the fundamental and related work. Fundamental work introduces the tools, techniques, and approaches we use as the basis of this thesis work. The related work includes other approaches and tools that also targeted at automating experimentation. Our work improves and complements the related work.

In Chapter 3 we provide some illustrative examples of the real world scenarios that our trial automation framework may get used. Two of the examples are described in detail to help the explanations of the framework in the following chapters.

Workload generation, the first step of the trial automation framework, is introduced in Chapter 4. Workload generation is to generate workloads that can capture the typical system usage or environment variation scenarios. We thus first discuss our approach to characterize the distributed system usage scenarios and distributed environment variation scenarios by classifying them into four levels. Then, we introduce our simulation-based workload generation approach. Through case studies, we discuss and illustrate that this approach has the ability to model the actor behavior of either of the four levels.

Upon the readiness of workloads, our model-based generative approach takes over the following experiment trial process. The process can be divided into five main phases. The first phase, configuration modeling, which is presented in Chapter 5, is the only phase that a software engineer needs to interact with. All the following phases presented in Chapter 6 are automated.

Reliability support is another important issue in the trial automation framework to support experiments on unreliable wide-area networks. In Chapter 7, we discuss the two reliability mechanisms we designed, recovery mechanism and bypassing mechanism. Both of them aim at correctly fixing an experiment trial, at the same time trying to maximize reuse and avoid repetition.

All our hypotheses are validated in Chapter 8 through case studies and meaningful experiments. Our evaluation is separated into five parts to answer five research questions. We discuss each of them in a separate section.

Chapter 9 concludes the thesis and explores some interesting future research directions that originate from this thesis work.

# Chapter 2

# Fundamental and Related Work

## 2.1 Fundamental Work

Our research is founded on many research areas, including discrete-event simulation [58], generative programming [16], distributed testbed [50, 61, 57], software deployment and maintenance [25, 35, 48, 4, 29, 30], parallel processing [12], and distributed monitoring [43]. Actually, development in these research areas motivated us to further explore the possibility to provide a consistent, unifed experiment automation framework.

### 2.1.1 Discrete Event Simulation

Discrete event simulation is a simulation method used to model real world problems that are able to be decomposed into a set of logically separate processes that autonomously progressing through time. The state of the simulation are assumed to change only at discrete points in simulation time. Besides simulation processes, the model of a discrete event simulation also consists of events. Each event takes place in a specific process at a logical time (a timestamp) and results in a state change of some process. The execution of an event may change state of the simulation, or schedule a future event, or pass a message to one or more other processes. On arrival at the other processes, the content of the message may result in the generation of new events, to be processed at some specified future logical time [58].

As an example, SSim[1] is a utility library that implements a simple discrete-event simulator both in C++ and in Java. A simulation programmed with this library consists one or more processes, each one executing a number of events, at given intervals, or in response to signals from other processes. The library defines the basic interface of a process and the main simulation scheduler. The simulation scheduler provides the methods for creating, starting, and stopping processes, for scheduling events, and for signaling other processes.

Discrete event simulation has been widely used in different areas [41]. Since it is well suited to describe various processes running concurrently with stochastic event arrival patterns, it can be well adopted to simulate distributed systems to evaluate their performance.

Alexander Egyed [21] proposed an approach to intertwine the model world and the real world through dynamic simulation. In dynamic simulation, simulating components may interact with real, un-verified, executing components within un-verified environments. Dynamic simulation is therefore a complement to validation and testing under situations where it is uneconomical or infeasible to model un-verified components. Mediators and translators were used to ensure that neither model nor code needed to be tailored towards dynamic simulation scenarios.

In our work, instead of modeling distributed systems, we adopt the discrete event simulation to model the actor behavior in using or interrupting the distributed systems, with each actor as an individual simulation process changing their behavior at discrete points in simulation time in response to some events.

### 2.1.2 Generative Programming

Generative Programming (GP) is "a software engineering paradigm based on modeling software system families such that, given a particular requirement specifica-

[1] http://www.cs.colorado.edu/ carzanig/ssim/index.html

tion, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge. [16]" Instead of building from scratch, it generates specific software systems based on a common generative domain model for the family of these systems. The model has three elements:

- Problem space: a means of specifying family members;
- Solution space: elementary, reusable implementation components;
- Configuration knowledge: mapping between a requirement specification of a member and a customized and optimized member.

Because of the reusability of the solution space and the configuration knowledge, generative programming provides efficient cost savings in repeated software development and provides better software quality than hand-written software implementation [16].

The generative programming paradigm provides us a new way of thinking. There exists a great amount of repetition in experiment script programming for different distributed system experiments. These scripts operate as members of a family sharing the common generative domain model for distributed system experiments. If we have the generative domain model available, the process to produce concrete scripts for a specific experiment is automated and straightforward through a generator.

Generator can be implemented in different ways, including the template-based generation, transformation-based generation, metaprogramming capabilities built into a programming language, and modularly extendible programming environments based on active libraries. We discuss two of them in detail.

**Template-Based Generation**

Most of current generators have templates and meta data as their inputs. Meta data are used to describe the characteristics about the program to be generated. Generators can generate target files by manipulating the meta data according to the templates.

There are some well-known template languages (TL), such as XSLT[2] for XML-encoded data. XML excels at representing structured data. It is portable and lightweight. Besides, many generators also design their own template language syntax. Some example generators (FreeMarker[3] , Velocity[4] , Jostraca[5] , Jamon[6] , TL[7] , MetaL[8] ) are listed in Table 2.1.

| **Generator** | **TL** | **Meta Data** | **Target** |
|---|---|---|---|
| **XSLT** | XSLT | XML | textual document |
| **FreeMarker** | FTL | Java | textual document |
| **Velocity** | VTL | Java | web pages, SQL, PostScript, etc. |
| **Jostraca** | Java Server Pages syntax | Java, Perl, Pathon, Ruby, Rebol, and C | code |
| **Jamon** | Jamon Template Language | Java | dynamic HTML, XML or text output |
| **TL** | TL | Java | code |
| **MetaL** | XML | XML | PHP, Java, Perl |

Table 2.1: Generators

### Transformation-Based Generation

Macro expansion as an example of transformation-based generation is a tricky operation in which a set of keystrokes and instructions are recorded, saved, and assigend to a short key code; when the key code is typed, the recorded keystrokes and instructions are executed.

Autoconf[9] and Automake[10] are both applications of GNU m4[1] macros. Autoconf is to provide a framework for handling the portability of source codes among different Unix variants. GNU m4 is used for Autoconf to generate Bourne shell configure scripts from template file **configure.in**. **configure.in** lists all the platform features

[2] http://www.w3.org/TR/xslt
[3] http://freemarker.sourceforge.net/index.html
[4] http://jakarta.apache.org/velocity/index.html
[5] http://www.jostraca.org/
[6] http://www.jamon.org/
[7] http://www.program-transformation.org/twiki/bin/view/Transform/TL
[8] http://www.meta-language.net/
[9] http://www.gnu.org/software/autoconf/
[10] http://www.gnu.org/software/automake/

the package will use in the form of m4 macros. The generated scripts run feature tests to determine the capabilities of the platform and configure the source code package. Although autoconf has solved the difference between different Unix variations, a complex **Makefile.in** file still needs to be written to describe source files and building rules. Since most programs are built in much the same way, there is a great deal of duplication in writing **Makefile.in** files, automake is developed in which GNU m4 is used to generate **Makefile.in** file from a much simpler user-provided file **Makefile.am**. The **Makefile.am** file need only describe the files used to build a package. Automake automatically adds the necessary rules when it generates the **Makefile.in** file. It also adds any rules required by the GNU **Makefile** standards.

Jatha[11] is a preprocessor that runs arbitrary Java code in order to generate the source code that the compiler sees. Jatha macros are classes in a Java package named "macros". All of them have a method named **expand()** that will turn an argument array into the source code for a property with **get** and **set** methods.

### 2.1.3 Distributed Testbeds

As one of the main elements of an experiment, a testbed supports an experiment by providing the experiment execution environment. This thesis is targeted at distributed systems, which have been widely used in local networks, wide-area networks, and the Internet. Thus, the testbeds used in real experiments with distributed systems should provide similar environments to the real networks. In this section, we introduce some actively used testbeds for distributed systems. Our experimentation framework is adaptable to any of them. These testbeds can be sorted to three categories, local-area networks (LAN), wide-area overlay networks (WAN), and emulated networks.

**Local-Area Networks**

The most easily accessed testbeds are local-area networks. Software engineers nor-

[11] http://www.kimbly.com/code/jatha/

mally take advantage of the networks administrated in their organizations to evaluate their developed systems. The advantage of using a local-area network is its flexibility. Software engineers can freely change the network configuration at their own will. And when an error happens, they can quickly examine and fix the errors. Because of the small-scale geographical distribution and the high-speed connection of LANs, the communication in LANs is very fast and reliable. The fast speed avoids many unexpected network failures, which may affect the experiment results. However, the small-scale and the high-speed also bring a shortage of local network environments: they cannot capture the features of the wide-area networks and the Internet, which involve longer latency, more heterogeneity, and more errors. Thus, local-area networks are more suitable for experiments in the early stage of software evaluation, when the basic functionality of the system needs to be examined.

**Wide-Area Overlay Networks**

Obviously, experiments with distributed systems would greatly benefit from a large-scale or even planetary-scale testbed, in which machines are distributed over the globe to mimic the Internet environment. An experiment deployed on such a testbed can have multiple vantage points from which the SUE can receive and react to service calls from distributed points. The internal communication between the SUE components can be affected by the heterogeneity and dynamic features of the testbed. A distributed system can be distributed across multiple administrative boundaries similar to the environments that it is deployed in reality.

Currently, the PlanetLab [50] overlay network is actively developed, maintained and used. Its developers and users, including academic, industrial and government institutions, form the PlanetLab Consortium. They cooperate to support and enhance the PlanetLab overlay network. As shown in Figure 1.2, PlanetLab is widely distributed around the world. On Jun 22, 2006, it consisted of 683 nodes hosted by 330 sites spanning over 25 countries. All the nodes are connected to the Internet. The purpose of PlanetLab is to provide an open platform for developing, deploying, and accessing planetary-scale services under real-world conditions at large scale. The PlanetLab supports both short-term experiments and long-running network services.

Since services are expected to run continuously, PlanetLab is organized as an overlay network. Overlay networks use encapsulation to enable virtual infrastructure. The primary advantage of the overlay network architecture is that it does not require universal network support to be useful. This enables faster deployment of desired network functions and adds flexibility to the service infrastructure, as it allows the co-existence of multiple overlay networks each supporting a different set of service functions. In specific, each service acquires and runs in a “slice” of the PlanetLab overlay. Each slice is a collection of virtual machines running on one or more PlanetLab nodes.

The X-Bone [57] is a system for rapid, automated configuration, deployment and management of overlay networks. It uses a graphical interface and multipoint control channel to manage overlay deployment rapidly at the IP layer. By making overlay establishment a fast, common function, the X-Bone enables new uses for overlays, such as for distributed applications without cumbersome application-level service location and routing support.

The ABONE[12] is another overlay testbed. It allows service developers to dynamically load their applications onto the overlay’s nodes.

### Emulated Networks

Although large-scale testbeds provide realistic experiment environments, they cannot meet all the experimental needs. For example, an experiment is to evaluate the performance of the SUE when a set of machines suddenly get very slow. The properties of the machines and the topology cannot be easily modified in the large-scale network testbeds. One possible while unsatisfying way is to deploy an extra tool, such as a traffic generator, on the testbed to modify the network properties of the testbed. An easier

[12] http://www.isi.edu/abone/

way to meet such experiment needs is to use network emulation.

A network emulation[13] is a technique through which a local-area network acquires the properties of a wide-area network. It is usually accomplished through the insertion of traffic generators and protocol implementations partitioning the LAN such that they may alter the packet flow between the halves to simulate specific environments. They have been implemented in the past for a variety of purposes on a variety of systems. Among them there are Nist Net [9] and Dummynet [52].

A useful way to think of Nist Net is a "network-in-a-box", a specialized router which emulates statistically the behavior of an entire network in a single hop. Nist Net selectively applies network effects, such as delay, loss and jitter, to the traffic passing through it, based on user-supplied settings. Dummynet is a simple, yet flexible network emulator for bandwidth management and for testing networking protocols. It works by intercepting packets in their way through the protocol stack, and passing them through one or more pipes which simulate the effects of bandwidth limitations, propagation delays, bounded-size queues, packet losses, etc.

A widely used emulated network testbed for distributed systems, Emulab [61], is based on Dummynet. Emulab provides an integrated emulation and simulation environment where the user can use traffic-shaping techniques to configure the network cluster to have the desired network profile. Although Emulab is not largely distributed around the world, given a realistic network profile, Emulab can provide a realistic and more controllable network environment than PlanetLab. The network properties can be configured with a NS-2 [22] like file to set up an emulated network testbed. Besides, Emulab also supports dynamic changes to the emulated network testbed through its event system.

StarBED,[14] another similar emulated network testbed, also targets to provide

[13] http://en.wikipedia.org/wiki/Network_emulation
[14] http://www.starbed.org/

realistic environments to evaluate Internet technologies. There are 512 PCs in StarBED, while the VMWare[15] software enables ten virtual machines to run on each PC. The VLAN [34] configurable switches enable the configuration of the network properties. But it does not support dynamic changes to the network during experiment execution.

### 2.1.4 Software Deployment and Maintenance

Software deployment and maintenance is widely studied as a facet of configuration management [25, 36, 35, 48, 4, 29, 30]. It is a software framework for automatically installing, updating, reconfiguring, adapting, and removing software systems. Even for a distributed system, it allows system users to consider all the details of a software system in one place. This makes the configuration of the system easier and reusable. There are numerous commercial and public tools available that provide many advance features for software deployment (e.g., SmartFrog [25], Castanet [36], Tivoli,[16] InstallSheild [35], netDeploy [48], RPM [4], and the Software Dock [29, 30]). There are also many tools associated with distributed testbeds. Software deployment and maintenance provides a solid foundation for the experiment preparation step, in which the SUE and the experiment necessities need to be deployed onto a distributed testbed.

### 2.1.5 Parallel Processing

Parallel processing is a well-developed technology. It can greatly improves the efficiency in executing a set of independent commands. Because of the long network latency of PlanetLab, some parallel execution tools, such as Parallel SSH [12], vxargs,[17] plDist,[18] and CoDeploy,[19] have been popularly used by its users. They archived faster file deployment and command execution through parallel communication.

[15] http://www.vmware.com/
[16] http://www-3.ibm.com/software/tivoli/
[17] http://dharma.cis.upenn.edu/planetlab/vxargs/
[18] http://www.arl.wustl.edu/∼mgeorg/plDist.html
[19] http://codeen.cs.princeton.edu/codeploy/

## 2.2 Related Work

This section discusses the current state of experimental approaches, techniques, and tools applicable to distributed systems. Their functions are also clarified by being fit into the three-step process for each experiment. We separate the related work to three parts: workload generation, experiment management, and data collection and analysis.

### 2.2.1 Workload Generation

Synthetic workload generation [7, 66, 6, 31, 20] studies how to generate realistic inputs (workload) to an experiment. In general, three main approaches have been used to synthesize a workload: trace-based approach, analytical approach, and operational approach.

#### 2.2.1.1 Trace-Based Approach

The trace-based approach starts with an empirical trace and either subsamples it or permutes the ordering of the system requests in the trace to generate a new workload different in some respect from the original. Many commercial software testing tools, such as DieselTest, [20] LoadSim, [21] and Eggplant, [22] use this approach with the support of some capture tools like Muffin.[23] Unfortunately, the trace-based approach has limited flexibility since the generated workload is inherent to a particular configuration of a specific system.

This approach is still commonly used nowadays for testing tools with the support of some capture tools. The captured traces are normally combined with scripting languages to generate test scripts.

For example, DieselTest[24] is a load injector to test Internet web sites under Win-

[20] http://www.dieseltest.com
[21] http://www.openware.org/loadsim/
[22] http://www.redstonesoftware.com/
[23] http://muffin.doit.org/
[24] http://www.dieseltest.com

dows NT environment. The scenario of a test is recorded automatically through an integrated capture tool in a browser by recording the links visited together with the content and the moment each of them is visited. It is also possible to edit the recorded scenarios.

LoadSim[25] is designed for HTTP distributed load testing. It uses Muffin[26] for creating scripts. Muffin captures the requests made in a browser. A filter provided with LoadSim allows Muffin to generate XML file with all the user requests on the web application.

Eggplant[27] is a commercial system for performing user-level testing on any host connected to the network. Interestingly, while Eggplant does not explicitly support any software deployment or installation of the system under test, its primary focus is on driving tests by replaying scripted user actions from a remote machine.

#### 2.2.1.2 Analytical Approach

In the analytical approach, mathematical models are created for the workload's factors of interest by fitting statistics derived from a set of traces to some widely-used distributions. Then, data are generated randomly to produce workloads that statistically conform to these models. Research on workload generation has typically focused on this approach [7, 66, 6].

Analytical approach could be further divided into resource-oriented approach and user-oriented approach. With the resource-oriented approach, the global characteristics of the system workload are modeled directly. ProWGen [7] is a synthetic web proxy workload generator that uses this approach. It models aggregate client workloads seen by a typical proxy server, rather than individual client. A work at RPI [66] also used this approach. Based on some well-known distribution models, it generates network

[25] http://www.openware.org/loadsim/
[26] http://muffin.doit.org/
[27] http://www.redstonesoftware.com/

traffic workloads for NS-2[28] simulations of wide area networks and the Internet traffic workloads for various network applications. In contrast, a more popular approach, the user-oriented approach, models each user's activities separately and extracts a set of workload actions from these models through mathematical deduction. Many researchers have been doing research using this approach [6, 47]. For example, Surge [6] is a workload generator designed to capture various traffic characteristics of web sites. Each user is modeled through a simple ON/OFF process called a User Equivalent (UE). Each UE alternates between issuing requests and being idle, exhibiting distributional and correlational properties characteristic of real web users.

The analytical approach's process from collecting traces, analyzing traces, to finally generating and parameterizing models normally can take up to several years depending on the accuracy to achieve [39]. This is in conflict with our vision to quickly get experiment necessaries ready. Thus, it is inappropriate to generate workloads for experimenting with pivotal systems, which usually handle emergencies such as worm propagation and real-time stock market tracking. Besides, analytical approach is not generally applicable to any workload scenarios. Most of the existing mathematical models [5, 28] are based on a small set of traces collected from one particular part of a network within some particular time period. Thus, they cannot be widely used for different system usage scenarios. Specifically, the analytical approach does not allow the SUE to be subject to hypothetical or anticipated usage scenarios since all the mathematical models are based on real application of the system. However, analytical approaches, which have been widely used for some applications such as web servers, still have their advantages. They provide concise statistical models. As long as the models are available, the process to generate a workload is fast.

[28] http://www.isi.edu/nsnam/ns/

#### 2.2.1.3 Operational Approach

Most of the available workload generators are focused on web server workloads. This is because of the widely usage of web-based applications. Since the currently dominant workload generation approach is analytical approach, which is based on the statistical models drawn from a large number of real execution traces, only the widely used web-based applications can generate that many real execution traces. For the relatively new distributed architectures, analytical approach obviously cannot meet the requirements to quickly get their workloads ready.

The operational approach, uses operational models of each system user's behavior to represent the workload characteristics of interest. Some of the models adopted include finite state machines, Markov chains, and Turing-complete programs. Through execution of the operational models, the workload is generated. Research on the operational approach, such as BISANTE [31], SynRGen [20], and SPECjAppServer, [29] is still in the early stages. Most of the applied models only capture simplistic scenarios. However, the idea of the operational approach provides great potential to quickly generate realistic workloads. Besides real execution traces, sources of operational models could also be some detailed user operational profiles [46], or the experimention plan in which the test scenarios are described, etc. Since software engineers have more experience with the operational model, they are able to generate complicated scenarios with the operational approach more easily than with the analytical approach. This is very valuable for providing the unified experimentation framework for a variety of distributed systems. Its flexibility and efficiency is very valuable for unexpected events and for the scenario different substantially from that of any currently available environments.

A research group at the University of Vienna examined an approach whereby the behavior of a single user was modeled with a hierarchy of independent stochastic processes exchanging standardized messages [31]. Although a user is modeled by several

[29] http://www.specbench.org/osg/jAppServer/

communicating processes, each user is independent from the others and from the system. Thus it cannot model dynamic client behaviors.

SynRGen [20] uses micromodels to capture the file access pattern of users. Complex behavior models are achieved by combining the micromodels stochastically. A micromodel is simply a finite state machine in which states represent distinct user behaviors, while transitions represent a user changing from one behavior to another.

SPECjAppServer[30] emulates an automobile dealership, manufacturing, supply chain management (SCM) and order/inventory system to drive the experiments that must be deployed and configured manually. It was designed to test the performance of J2EE applications.

QARun[31] seems to be a mixture of the trace-based approach and the operational approach. Its function is to capture a user's interactions with a system and store them in test scripts. Checks and event handlers can be added to extend these scripts. Checks serve to validate the behavior and performance of the system, while event handlers serve to handle any event from the system or wait for an indeterminate amount of time for an event.

### 2.2.2 Experimentation Automation

Our work to automate the experiments with distributed systems does not start from scratch. Because of the difficulties in controlling experiments on a large number of machines manually, software engineers have developed some techniques to promote their experimentation process. We separate these techniques into different types: prototype-based technique, distributed controlling technique, service reuse technique, configuration technique, testbed technique, fault injection technique, and experiment distribution technique. These techniques are discussed next.

[30] http://www.specbench.org/osg/jAppServer/
[31] http://www.compuware.com/products/qacenter/qarun.htm

#### 2.2.2.1 Prototype-Based Technique

In practice, prototypes are built early in the development process of distributed applications to initially assure that the software and hardware infrastructures that they use can provide the required levels of performance. Special experimentation frameworks have been developed to make this possible. Based on the users' descriptions about a distributed application's architecture, a prototype for the application is generated and put into experimentation. A distributed application is implemented based on lower-level distributed components, including middlewares, web servers, databases, etc. Its prototype is actually a test scenario for the lower-level distributed techniques.

For example, Grundy, Cai, and Liu [26] discuss the SoftArch/MTE system. This system enables automatic generation of the prototype for a distributed application together with its dedicated testbed based on high-level architectural objectives. The generated system is then deployed and the performance metrics are gathered while the system is exercised. Although it implements the function of generating test scripts based on the prototype configurations, the configurations are too specific as it is used to indicate client requests, server services, and particular kinds of middleware and database technologies to be used. Thus, this work can only be used to evaluate the design of client/server style distributed applications. In contrast, our aim in this work is more general and more focused on highly distributed systems.

Woodside and Schramm [63] pointed out the importance of studying the performance issues through experimentation while still planning the system. Instead of the implemented system, a synthetic representation is used in the planning stage. They present a tool called the Layered System Generator (LSG) which is used to create synthetic client/server style distributed systems. A synthetic task system can be used to generate network and workstation traffic which represents the load for a planned software system. Then the synthetic distributed systems could be executed to evaluate the planned software system. This work has the similar limitation as the SoftArch/MTE system.

Instead of only considering experimenting with distributed systems in real environments, Urb́an, Défago, and Schiper pointed out that the three basic approaches to evaluate the performance of algorithms, formal method, simulation and measurement, have their respective advantages and limitations. In order to increase the credibility and the accuracy of performance evaluation, it is considered good practice to compare results obtained using at least two different approaches. To remove the obstacle because of the fact that one ususally has to develop different implementation for different evaluation approach, they developed Neko [59], a single communication platform that supports both distributed algorithm simulation and execution on a real network, using the same implementation for the algorithm. The architecture of Neko consists of two main parts: application and networks. In the application part, the activities of different types of application processes are programmed in Java classes. The instantiated processes may communicate with each other. The application is configured by a single file, in which a real experiment or a simulation is described. The difference between the two types of configuration is that they point to different networks, either real or simulated networks. Neko then launches the application differently based on the configuration. For simulation, only one Java Virtual Machine (JVM) is launched. For real experiment, one JVM is launched for each process, all of which constitute a control network. The network and the application get initialized with the modules specified in the configuration file. The application begins to execute their programmed activities once the initialization finishes. A `shutdown` function is pre-defined in Neko to shut down all the processes after all the processes finish executing.

Different from the previous two tools, Neko is targeted at a variety of distributed algorithms other than the client/server applications. The communication between the application processes can form very complicated distributed algorithms and protocols.

However, Neko still only supports the experimentation activity early in the software development process. The programmed application in the Neko environment cannot be used independently as the real distributed system implementation. The real implementation needs further evaluation supported by other frameworks like this thesis work. Because of this, it mainly focuses on the performance of the distributed algorithm itself without providing the mechanism to model how the algorithm would be used.

#### 2.2.2.2 Distributed Controlling Technique

Distributed experimentation describes the ability to create and run experiments that have elements operating on multiple machines. However, the term "distributed" means not only a simultaneous run of experiment parts on a number of machines, which is specific for simultaneous testing; rather, distributed testing also suggests that experiment parts interact with each other during an experiment run. Such a scenario allows us to perform complex, synchronized and comprehensive experiment with client/server or highly distributed systems. A natural way to conduct experiments with distributed systems is to use a controller controlling the distributed experiment parts. Thus, a lot of tools have implemented a framework to control distributed execution of experiments.

**Client/Server System**

Because of the popularity of web services with client/server architecture, there exist many experimentation frameworks and benchmarking tools supporting the testing of client/server system, such as Coyote [64], Grinder,[32] LoadSim,[33] Jmeter,[34] OpenSTA,[35] LoadTest,[36] WebStone,[37] SPECweb99,[38] and Rubis.[39] They all have similar architectures in which test scripts are created for some test scenarios; then virtual clients

[32] http://grinder.sourceforge.net/
[33] http://www.openware.org/loadsim/
[34] http://jakarta.apache.org/jmeter/index.html
[35] http://www.opensta.org/
[36] http://www-svca.mercuryinteractive.com/products/loadrunner
[37] http://www.mindcraft.com/webstone/
[38] http://www.specbench.org/osg/web99/
[39] http://rubis.objectweb.org/

execute the test scripts at the target web services; the execution is monitored or logged with the results collected at the end of the execution. Most of the frameworks consider each SUE server to be independent from one another and consider each SUE client as an experiment entity interacting with a server from a remote machine during experiment. Thus, the measurement also takes the network communication time between clients and servers into consideration. Three approaches have been used by these products to create experiment scenarios:

(1) Those that allow the experimenter to write entirely his or her scenario, either directly using a scripting language such as Jython or indirectly generated from user-provided configurations, like Grinder, LoadSim, Jmeter, OpenSTA, WebStone, SPECweb99, Rubis, and Coyote.

(2) Those that allow the experimenter to build his or her scenario with some graphical objects representing built-in interactions, like Jmeter and LoadTest.

(3) Those that automatically record a scenario with a capture tool often integrated in an Internet browser, like DieselTest, OpenSTA, LoadSim, and LoadTest.

The execution step is similar for every product. One thread is created for each virtual user. A central controller controls all the virtual users by synchronizing and scheduling the creation and execution of all the virtual users.

In these products, the data is collected through monitors. They monitor the target system's main characteristics: number of virtual users, average response time, etc. But only LoadTest and Rubis provide very detailed information on the target system: number of processes, processor characteristics, memory usage, etc.

**Highly Distributed System (HDS)**

Many tools implement a typical architecture for experimenting with highly distributed systems, such as DECALS [33], Distributed TETware, [40] RiOT [55], and

[40] http://tetworks.opengroup.org/Products/distributed_tetware.htm

TestZilla.[41] An HDS is made up of a few components, all of which interact and contribute towards an aggregated experiment result. Its experiment is normally controlled by an "experiment controller", which communicates with all the components and controls their running from the beginning to the end and collects data from them. The process is driven by a workload that lists all of the actions and describes how they are processed. Different from clients in client/server systems, clients in HDS are not separate experiment entities in their experimentation framework. They are located on the same machines as their system access points. This thesis work also uses this central controller architecture.

#### 2.2.2.3 Service Reuse Technique

STAF (the Software Testing Automation Framework) [42] is a multiplatform, multilanguage approach to improve reusability and automation in the software experiment cycle as shown in Figure 1.1. In specific, reusability is focused on the preparation step, and to a less extent the design step of the experiment cycle. Automation is focused on the execution step. However, STAF is not targeted at distributed systems.

To improve reusability, STAF is designed around the idea of reusable components called **service**s. The three core services in STAF are the handle, variable, and queue services. Based on them, services supporting synchronization, process execution, file system, logging, remote monitoring, and event-handling are all implemented in STAF. For example, process execution facilities allow processes on STAF to be started, stopped, and queried. File system facilities support file transfer and access. In addition, users could always develop their own services to meet their specific needs. Automation is implemented through leveraging the above services. STAF itself is fundamentally a daemon process that provides a thin dispatching mechanism that routes incoming

[41] http://www.cs.cornell.edu/vogels/TestZilla/default.htm
[42] http://staf.sourceforge.net/index.php

requests to these services.

#### 2.2.2.4 Configuration Technique

Some testing tools do provide flexible configuration mechanisms for constructing different experiment scripts automatically, but their configuration models are fairly narrow in scope. They are targeted at specific architectures rather than generally applicable to different types of distributed systems.

The JXTA Distributed Framework (JDF)[43] is an automated distributed functional testing framework for JXTA J2SE 2.0 platform[44] in order to build and run a distributed test more easily and systematically. It focuses on configurations of JXTA networks. JDF provides a set of Unix shell scripts that use Java components for the automatic creation, configuration, bootstrap and result retrieval of a network of JXTA peers across multiple Unix hosts given an XML JXTA network descriptor. The current implementation uses the SSH protocol to access remote hosts, while JXTA-based mechanism is expected to replace the SSH to allow for better control over peers and compatibility on Windows platforms.

CCMPerf [38] is a benchmarking framework to compare CCM (CORBA Component Model) implementation quality by developing metrics that evaluate the suitability of those implementations for the representative distributed real-time and embedded systems. Design of CCMPerf solves the heterogeneity problem in CCM implementation by generating platform-specific code and build files from a set of configuration and execution scripts; differences in quality of CCM implementations by a combination of white-box and black-box metrics for evaluation; differences in CCM configuration options by scripts to configure and run each test automatically; and differences in application domain via scenario-based test of specific use cases deemed important in a given

[43] http://jdf.jxta.org
[44] http://platform.jxta.org

domain. CCMPerf supports formal experiments with CCM implementation in which a set of trials need to be done to get a range of black-box and white-box metrics for each benchmarking test.

#### 2.2.2.5 Testbed Technique

In Section 2.1.3, we introduced a number of distributed testbeds for public use [50, 61, 60]. Any one of them is able to handle deployment of user-provided network-level configurations when an experimenter sets up an experiment testbed. When the testbed is set up and ready for use, it becomes difficult to control the distributed programs on many machines manually. The experimenter has to log into each experiment node and type commands at the scheduled times. Moreover, human operations are not accurate, and cause the decreasing of the experiment reliability. Thus, to streamline their usage, maintainers and users of these testbeds have contributed a number of tools and services for the engineering activities on the testbeds.

**PlanetLab**

PlanetLab [50] is a popular planetary-scale testbed for distributed systems. To use PlanetLab, an engineer interacts with each of the hosts associated with their slices using Secure Shell (SSH)[45] to perform necessary configuration and execution. A few tools are available to assist the engineer in using PlanetLab, including the Nixes Tool Set,[46] the PlanetLab Application Manager,[47] and Stork.[48] All of them help install, maintain, monitor and control an application installed on Planetlab.

The Nixes Tool Set provides a set of bash scripts to install, maintain, control and monitor applications on PlanetLab. It bootstraps the nodes with yum and lets the user install rpms from the PlanetLab distribution. Nixes was designed with simplicity and performance in mind. The tool is based on tree components: the scripts, a configuration

[45] http://www.openssh.com
[46] http://www.aqualab.cs.northwestern.edu/nixes.html
[47] http://appmanager.berkeley.intel-research.net/
[48] http://www.cs.arizona.edu/stork

file and a repository (public web directory) hosting the application. It is focused on pushing and maintaining software on large numbers of similar hosts and parallelizing the execution of shell commands.

The PlanetLab Application Manager appears to be a more robust tool that allows periodic updates to be pulled from a server-based repository. But it is still designed to help centrally manage, install, upgrade, start, stop, and monitor applications on PlanetLab.

With the same goal to help installing and maintaining applications on PlanetLab, Stork provides extra benefits by intelligently optimizing the application installation traffic through the techniques of caching, and package-sharing between slices.

All of these three tools are targeted at application deployment without concern for experimentation with the application. Thus, they do not support descriptions of different experiment scenarios. However, all of them could be utilized as underlying technologies for our thesis work.

**Emulab**

Another widely used testbed for distributed systems, Emulab [61], provides an integrated emulation and simulation environment where the user can use traffic-shaping techniques to configure the cluster to have the desired profile. The Emulab management system provides ways for the engineer to distribute software onto the hosts used in their experiment. While there is some support for experiment management, including a batch system for running experiments without user intervention, [49] there is, similar to PlanetLab, no support for describing different experiment scenarios specified in the workloads and the experiment configuration in our thesis work.

An automated regression testing framework, DART [13], has been implemented targeted at wide-area distributed applications in the Emulab network environment. It provides a set of primitives for the engineer in writing distributed tests and a runtime

[49] http://www.emulab.net/tutorial/docwrapper.php3?docname=tutorial.html

that executes distributed tests in a fast and efficient manner over a network of nodes. But no automated workload generation or script construction is supported by DART.

ACME [49], a framework for automated robustness evaluation of distributed services, targets both emulated network testbeds such as Emulab as well as real wide-area testbeds such as PlanetLab. The workloads in ACME are actually perturbations to introduce failures to the SUE and the testbed, such as component dying and high network usage. These perturbations are controlled, monitored and injected through per-node sensors and actuators. ACME only supports robustness experiments since it provides no control over service calls. Moreover, the same as all the other experiment frameworks we mentioned, it does not consider automated workload generation or script construction.

**X-Bone**

The X-Bone has an extension that explicitly deals with application deployment [60]. With this extension, a software engineer can request a virtual network with certain topology and also specify the applications to deploy over it. Once the network nodes are found, a user-provided application generator script is instantiated with the overlay-specific parameters to generate an action file generator script. The action file generator script is modified again with the node-specific parameters to generate a node action file. The node action file is in charge of configuring, starting, monitoring, stopping and cleaning up the node.

**StarBED**

The StarBED[50] developers developed an experiment software **SpringOS** [45] to support the experimenters in operating experiments in the StarBSD environment. It only requires an experimenter to prepare an experiment configuration file, in which the node configurations, the network topology, the global scenario and some node scenarios are described. The **global scenario** is a list of commands or execution steps of the

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

scenario master, while each **node scenario** is for a testbed node.

Based on the configuration file, SpringOS executes experiments automatically. It first acquires the experiment nodes following the criteria described in the configuration file, sets up the nodes by installing different disk images specified in the configuration file, and builds up the experiment topology by means of VLAN [34]. Then it copies the node scenarios to the experiment nodes and starts the experiment. A **scenario slave** is deployed on each node to execute the scenario independently. Only when the nodes need to synchronize, the scenario master will mediate among the scenario slaves. The whole process is accomplished with a list of modules constituting SpringOS.

SpringOS shares a similar architecture characteristic to our thesis work in that the scenario master coordinates the execution of the distributed scenario slaves. It also shows some levels of flexibility by supporting the scenario definition in which a node may take different actions according to the received messages. It implements this feature through the `msgswitch` statement in the configuration file. However, this software is still limited in that it is specific to StarBED testbed. The experimenter uses this software by directly providing testbed-specific scenario descriptions. It cannot be easily applied to other distributed testbeds. It considers each client to a server as a separate entity located on a different testbed host. Thus it is more proper for experiments with client/server systems instead of distributed systems we study in this thesis. Neither does it support a variety of experiment scenarios like robustness testing.

#### 2.2.2.6 Fault Injection Technique

One possible experimental goal of a software engineer is to validate the functionality of a system under failure scenarios. Such experiment, called dependability evaluation or robustness testing, is especially important for distributed systems since they are widely deployed on the unreliable, uncontrollable wide-area network environments. However, it is also particularly difficult for distributed systems since failure scenarios may occur due to a correlated occurance of faults in a distributed way. Many fault injection techniques and tools have been developed to help software engineers easily receate failure scenarios and exercise the subject system in the scenarios [32].

One challenge in re-constructing a failure scenario for a distributed system is to decide when to inject faults based on its global execution states. Cukier et. al. addressed this issue with a fault injector, Loki [11]. In an experiment, Loki is deployed to form a framework for maintaining a partial view of the global state, and for conducting fault injections based on that partial view. The partial view is derived using state machines associated with communicating Loki nodes. The inter-node communication does not block the subject system, thus the partial view of the global state seen by Loki is not always correct. Loki handles such problem by reviewing the fault injection actions after the experiment is completed, to see whether the faults were injected as intended.

The research on fault injection techniques for distributed systems does not focus on experiment management, but instead on failure scenario modeling. This thesis work can also model failure scenario using the environment actors. Our trial automation framework provides the environment to apply failure scenarios on a distributed system. It also provides the basic support for actors to communication partial states to form a global state of the subject system. However, since the state of the distributed system is not our focus, this thesis work does not cover the algorithms and mechanisms used in the inter-actor communication to study the system's states.

#### 2.2.2.7 Experiment Distribution Technique

Besides automating the sequential experimentation process, another intelligent technique to promote the experimentataion activity is to distribute an experiment to distributed sub-experiments that run in parallel. This technique shortens the length of an experiment by increasing its breadth.

Skoll [44] supports "distributed quality assurance (QA)" by leveraging the extensive computing resources of worldwide user communities in a distributed, continuous manner to significantly and rapidly improve software quality. It distributes a QA (Quality Assurance) task to distributed subtasks. Skoll is based on a client/server model, in which clients request job configurations (QA subtask scripts) from a server that determines which subtask to allocate, bundles up all necessary scripts and sends them to the client. This approach allows applications with a large space of possible configurations to be tested in parallel. Although Skoll provides an infrastructure for performing large-scale QA, the configuration spaces of performance-intensive software systems are often so large that brute-force processes are still infeasible. Thus, the Skoll developers enhanced the Skoll environment with a model-based approach that enables it to rapidly identify a small subset of highly influential performance-related configuration options and systematically explore that subset of options empirically to estimate system performance across the entire configuration space [65]. Skoll has been applied to evaluate QoS-enabled middlewares used in distributed real-time and embedded (DRE) systems [37]. Although Skoll has the similar goal of promoting the experimentation activity in software engineering, it is targeted at the experiment-level automation instead of the trial-level automation, which is our focus in this thesis work.

### 2.2.3 Data Collection and Analysis

We take a simple off-line approach for data collection and analysis in this thesis work, through which customized post-processing scripts are deployed and executed against the trial execution log files, the resulted diagnostic data are collected back to the engineer for further analysis. This simple approach worked for all the experiments we conducted using Weevil. But more detailed analysis of the distributed systems would require more advanced techniques in handling the large quantities of experimental data. Many currently available techniques can be used for such purposes.

**Log Analysis** Instead of manually programming scripts to parse the log files, the normal practice to study the widely used web servers is to use the log analysis products. There exist a large number of log analyzers, such as AWStats [19], Http-analyze,[51] and Webalizer.[52] These tools are able to generate advanced statistics based on the log files of the web servers, the ftp servers or the mail servers in real time. However, almost all of them are only applicable to these several types of servers. For the other distributed systems, these tools cannot help.

**Distributed Data Analysis**

Compared with the analysis of each individual log file, improving the distributed data collection and analysis is more important. An experiment with a distributed system can generate a large number of events, recorded in the log files distributed on the wide-area testbed. Moreover, correlated events can be generated concurrently and distributed in various locations. Thus, some log files cannot be stripped off individually before they are analyzed as a whole.

To support such aggregated analysis while avoiding overloading the network and seriously affecting the system execution, distributed data analysis techniques have been studied in many distributed monitors to provide a scalable way for processing the events generated by distributed systems. They employed many distributed protocols and techniques, including the publish/subscribe event notification and the distributed data mining, to implement scalable distributed data aggregation.

Al-Shaer et.al. proposed a scalable, flexible and non-intrusive monitoring architecture for highly distributed systems [2]. The architecture is scalable through a hierarchical publish/subscribe event notification approach that distributes the monitoring load and limits event propagation. The users can flexibly express their monitoring demands via different subscriptions on-the-fly at run-time. The notification messages

[51] http://http-analyze.org/index.html
[52] http://www.webalizer.com/

come from the events generated by the subject system at run-time.

Astrolabe [51] is another scalable technology for distributed system monitoring. It also implements scalability through the hierarchical organization. Distributed resources are separated into hierarchical zones. From the lower zones to the higher zones, Astrolabe continuously computes the summaries of each of them using the on-the-fly aggregation mechanism. The mechanism is flexibly controlled by SQL queries. It allows users to install new SQL queries on the fly to customize the aggregation.

Ganglia [43] is a scalable distributed monitoring system especially for high-performance computing systems such as clusters and grids. It is based on a hierarchical design targeted at federations of clusters. It relies on a multicast-based listen/announce protocol to monitor state within clusters and uses a tree of point-to-point connections amongst representative cluster nodes to federate clusters and aggregate their state.

# Chapter 3

# Illustrative Examples

This thesis work is applicable to a variety of distributed systems on different testbeds to achieve wide-ranging experimental goals such as design strategy selection, algorithm adjusting, and dependency analysis [3]. Some typical experimentation activities include performance evaluation, functional testing, and robustness testing. Our trial automation framework treats the SUE as a collection of distributed self-contained components, generates trial scripts for each component, then manages and coordinates their execution.

For example, three levels of functional testing are normally considered in software engineering: the unit test, the integration test and the system test. The unit test is to test software units, like single classes or objects, isolated from the context they will act in. The unit test of a distributed system is to test correctness of the modules contained in a component. In contrast, the integration test and the system test for distributed systems execute test cases against integrated distributed components or the whole system. The execution outputs are checked against the test oracles. Our framework can help automate the integration and system test process for distributed systems in network environments.

In this chapter, we present three example scenarios that demonstrate the situations in which our thesis work can be applied. The experiments conducted in these three examples have different experimental goals. The first is to select a good design strategy to build a server-based service, the second is to test the functionality of a system, and the last is to adjust a distributed algorithm for better performance.

Phil, a software design engineer, is in charge of a project to construct an online banking system for a bank. His initial idea is to build this system with some web server products that he is familiar with. But he cannot decide which one to choose. He gets some potential usage scenarios from a bank manager, which give him some ideas about the user behaviors of the banking system. Phil is also provided the information about the network platform on which the system will be deployed. To compare several web server products, Phil wishes to collect some metrics about the products, such as their request latency and throughput. Phil generates some workloads based on the usage scenarios and sets up a local network that is similar to the network environment in the bank where the service will be deployed. Then he applies the workloads to perform experiments with each of the several web server products in the local network. The collected execution log files provide him the base to analyze and compare these products and decide which one to choose.

Paul, a software test engineer, has just finished unit testing of an implemented component-based distributed system. While he has comprehensively tested the correctness of each component, the target of this system is to support Internet applications. So he plans to conduct system testing in which a number of components are deployed on a large-scale testbed and executed cooperatively. To experiment with the system in realistic environments, he decides to use a testbed created on Planetlab. With the help of a test case generator, he gets a test suite made up of a list of test cases together with their oracles. Then he applies the test suite and check the execution results against the oracles.

Andrew, a researcher in distributed systems, has recently released the first version of a distributed service in which he implemented a distributed hash table algorithm. But he gets some complaints about the performance of the service under some execution environments. He has ever evaluated the algorithm through simulation before it was implemented, but the results did not show these problems. So he decides to experiment with the implemented service under the situations similar to those that the users described and adjust the parameters of the algorithm accordingly.

To be more specific, in the next four chapters, we refer to two example experiments to help illustrate different issues in our trial automation framework. The first experiment is aimed at studying the performance of web proxies under different caching policy parameters. Figure 3.1 shows a deployment diagram of the web proxy/web server

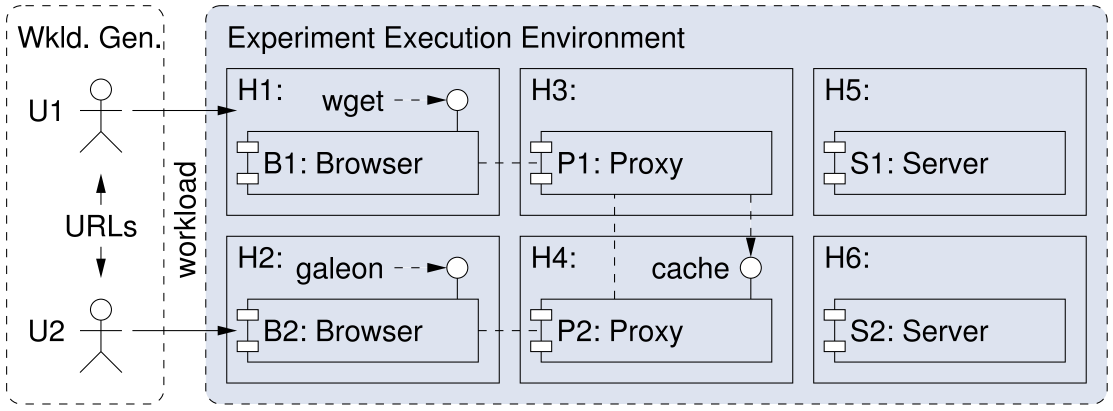


Figure 3.1: Deployment Diagram of Web-Proxy Experiment. In this experiment, two web browsers (B1, B2) are deployed on hosts H1, H2 of the testbed, two cooperative web proxies (P1, P2) are deployed on hosts H3, H4 of the testbed, and two web servers (S1, S2) are deployed on hosts H5, H6 of the testbed. Two clients (U1, U2) inject web access requestes to the SUE.

application on a six-host testbed used in this experiment. There are six SUE components mapped onto six hosts. The components are of three kinds: web browsers (B1, B2), web proxies (P1, P2), and web servers (S1, S2). 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 3.1 also shows two web clients (U1, U2) communicating with the SUE. The two clients exhibit an interdependent behavior, engaging in out-of-band communication (with respect to the SUE) by exchanging URLs.

The other experiment is aimed at testing the robustness of a Chord network. The Chord [54] network is a distributed system with the peer-to-peer architecture. Its main function is to efficiently locate the Chord node that stores a particular data item. Although the Chord binary distribution [1] requires at least 16 Chord (real or virtual) nodes to form a valid Chord network, we instead show a Chord network with six nodes in Figure 3.2 to simplify our discussions. In this experiment, there are six Chord nodes

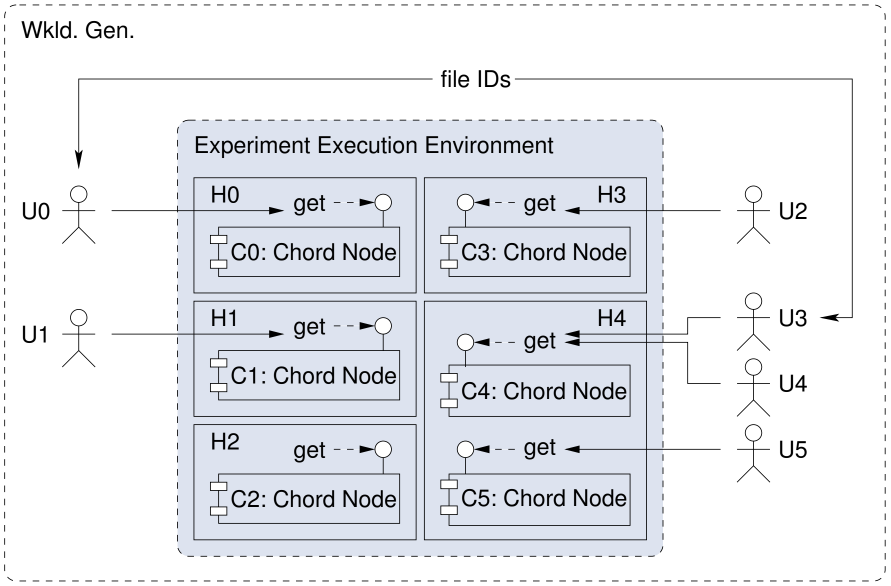


Figure 3.2: Deployment Diagram of Chord Experiment. Initially, six Chord nodes (C0-C5) are deployed on five hosts (H0-H4) of an Emulab testbed. Six Chord users (U0-U5) inject file access requests to the Chord system. The experiment execution environment changes during experiment execution.

(C0-C5) deployed on five hosts (H0-H4) initially, with node C4 and C5 both on host H4. Six Chord users (U0-U5) use the Chord network to access files through the Chord APIs. Besides, U0 and U3 are friends. Either of them may ask for help from the other if he or she fails to get response to certain requests from the Chord network. The system execution environment is unstable. Some Chord nodes, testbed hosts, or network links may fail and get fixed from time to time during system execution. In the following chapters, we demonstrate how our trial automation framework is able to conduct these

[1] http://pdos.csail.mit.edu/chord/

experiments automatically.

# Chapter 4

# Workload Generation

In this thesis, a workload stands for a sequence of actions performed in an experiment trial, such as service calls sent to the SUE, network failures happening to the system execution environment, etc. It is used to measure the effects on the SUE caused by these actions.

A distributed system is operated by a large number of distributed users and possibly affected by changes and failures in its distributed environment. We use the term **actor** to represent each user and each source of variations that may affect the distributed system. Actors actuate trial execution from distributed points. Two types of actors are considered in our framework, the **user actors** and the **environment actors**. The former actors actuate the trial through service calls to the SUE, while the latter through requests to change the SUE and the testbed. Their workloads are called **user workload** and **environment workload** respectively. Each of the two workloads lists all the actions performed by all the actors of a type. Each workload line specifies what time an actor performs an action.

The value of an experiment depends largely on the representativeness of its trial workloads. The ideal workloads are always successful in modeling some interesting real scenarios involving the SUE. They characterize the typical behaviors of the system users or the variations of the system execution environments. For example, stress testing and robustness testing of a distributed system are different types of experimentation. The workloads for stress testing model possible system usage scenarios in which the system users issue service calls in high intensity, while those for robustness testing model the possible failure scenarios of its network environment. As the extreme cases, real system execution traces can indicate workloads, but they are too limited in flexibility and availability. Thus, synthetic workloads that could capture the characteristics of the action sequences appearing in the real traces and logs are normally used in experiments. These generated workloads are flexible in modeling different scenarios by easily adjusting model parameters.

We consider workload generation as a separate step in our trial automation framework to generate a synthetic textual workload for each trial. Unlike the other steps to be discussed in Chapters 5 and 6, workload generation does not require the knowledge of the SUE or the trial execution information. In this step, a workload generator tries to pre-compute the actor behaviors as much as possible before a trial really executes. All the computable parts are saved in the workload with the incomputables left as variables in the workload to be resolved later during trial execution. We discuss the incomputables in Chapters 5 and 6. Such design can largely save the work in the trial execution phase to avoid the distractions on the trial results brought by the unnecessary trial-execution-time computing complexity.

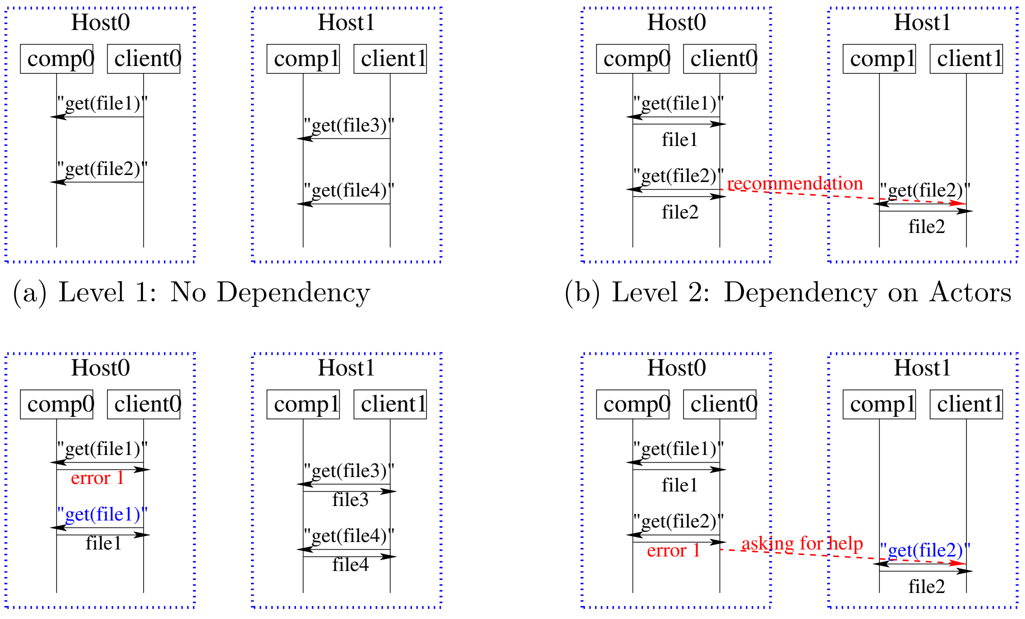
In each workload, we list the actions in an abstract, textual way to make the experimentation process of distributed systems more standard, cost-effective, and flexible than the popular practice of embedding actions in the test scripts. Using the textual format, the same workload can be applied to experiments with different distributed systems, with different real commands to implement the actions contained in the workload. In the web-proxy example shown in Figure 3.1, although the actions contained in the workload for the two web clients U1 and U2 are all presented as "GET"s, their corresponding real commands are different since they are accessing different web browser products, **wget** or **galeon**. The same workload can also be used in another experiment with a peer-to-peer file sharing system, where the "GET"s are translated to file retrieval requests. Like benchmarks, it is very likely that the **standard** workloads, i.e., the generally applicable workloads, can be widely distributed and used by software engineers to evaluate their systems under the same scenarios. Such reusability also makes the trial automation framework more **cost-effective** by saving the workload generation step. Besides such standard workloads, software engineers gain more **flexibility** in choosing to generate new workloads with the workload generators of their choice, or even choosing to use real traces or logs directly as long as the generated workloads have the compatible formats with that used in the trail automation framework.

## 4.1 Distributed Actor Behaviors

As illustrated in Section 2.2.1, current operational workload generation approaches only model the easily captured characteristics of actor behaviors. In specific, almost all of them only capture each actor as an independent workload process with routine behaviors. But in reality, a distributed system can be operated and affected by distributed actors displaying complicated, interdependent behaviors. Some behaviors may even depend on trial-execution-time information. In other words, they are not fully computable during workload generation step.

In this thesis, we attempt to model such dependent behaviors of the distributed actors as well as their routine behaviors. Because of the dependency, the actor behaviors of either of the user actors or of the environment actors are modeled by the combination of a workload and actor programs. The workloads are generated in the workload generation step as discussed in this chapter. Each workload line provides the textual descriptions of when, where, and how a request should be issued. A workload may not fully determine the actor behaviors in an experiment trial since some trial-execution-time variables are unsolved in the workload. Those variables will be solved in the actor programs when the programs analyze each workload line and translate it into a piece of code to actually issue requests in the trial execution phase.

We classify the actor behaviors into four levels based on their dependency on each other and on the trial-execution-time information. Figure 4.1 shows an example scenario for each level of actor behaviors. In Table 4.1, we summarize the elements we use to model each of them. Each column of the table shows an actor behavior level. Each row provides an element to help express the actor behaviors. The first element is the workload, while the next two are both implemented in the actor programs. The Xs in the table indicate all the elements we use to express the actor behaviors of each level.



(a) Level 1: No Dependency

(b) Level 2: Dependency on Actors

(c) Level 3: Dependency on Run-Time Info

(d) Level 4: Dependency on Both

Figure 4.1: Four Levels of Actor Behavior

The simplest scenarios involving actor behaviors of level 1 have been addressed by many workload generators. As shown in Figure 4.1(a), a user actor in this level keeps its pace in performing actions regardless of the system responses or the activities of other actors. Thus its behaviors can be fully determined in a workload as a list of routine service calls scheduled to be sent to the SUE. The user actor programs in this level just implement the straightforward translation from each abstract, textual workload

| Methods | Dependent behaviors | | | |
|---|---|---|---|---|
| | **None** **Level 1** | **On actors** **Level 2** | **On run-time info** **Level 3** | **Both** **Level 4** |
| **Workload** | X | X | X | X |
| **Execution-time info analysis** | - | - | X | X |
| **Execution-time communication** | - | - | - | X |

Table 4.1: Levels of Actor Behaviors

line to the real service calls using the system APIs. Similarly for an environment actor, its textual descriptions of the interruption to the system execution environment are translated into real commands using shell scripts, system APIs, testbed APIs, etc.

The actor behaviors of level 2 are dependent on other actor behaviors. The dependency may lie in the same actor, or between different actors. For example, a user actor may schedule to send out a service call that incorporates data from a previous service call. In the example scenario shown in Figure 4.1b, the dependency is between different user actors. The behaviors of client0 and client1 are dependent on each other. Client0 recommends files to client1 from time to time. Whenever client1 gets a recommendation notification from the other, it schedules a retrieve request for the file. Since the actor behaviors of level 2 are still independent of the trial execution, they can be pre-computed during workload generation, and only the translation from the workload to system requests needs to be implemented in the actor programs.

The trial execution gets involved in the next two levels of actor behaviors (level 3 and 4). Consider the scenario shown in Figure 4.1c: client0 continuously issues retrieve requests. And it may reissue the same requests if a request fails. In this scenario, whether a request fails depends on the real execution status during trial execution. Consider another scenario involving actor behaviors of level 3, for the web-proxy example (Figure 3.1): The client U1 randomly picks URLs on the two web servers (S1, S2) and sends out web access requests every 100 seconds. After every 1000 seconds, if the average request response time is less than two seconds, U1 will increase his or her request rate from 1 per 100 seconds to 1 per 50 seconds. In this scenario, the average request response time cannot be foreseen in the workload generation step, and thus needs to be solved in actor programs during trial execution. Besides the unconditional actions which will be performed anyway, a workload for the actor behaviors of level 3 also include conditional actions. Whether the conditional actions will be issued or not depends on the trial-execution-time information received and analyzed in the actor programs during trial execution.

The actor behaviors of level 4 depend not only on other actor behaviors but also on the trial execution. Shown in Figure 4.1(d), instead of reissuing the request again if a request fails, client U0 chooses to ask for help from client U1. Thus, client U1's behavior to retrieve file2 is dependent on client U0's resort behavior as well as on the system response during trial execution. Since the inter-actor dependent behaviors in this scenario are undetermined before trial execution, the trial-execution-time inter-actor communication support is necessary in addition to the other three methods. During trial execution, based on analyses of the SUE's responses, the actor can decide whether to communicate with another actor.

In this chapter, we only discuss how to generate the workload, i.e., the pre-computable part, to describe actor behaviors of the four levels. The actor programs will be discussed later in Chapters 5 and 6.

## 4.2 Simulation-Based Workload Generation

A common practice in software experimentation is to generate a workload on the basis of a statistical model that abstracts usage patterns of the SUE (discussed further in Section 2.2.1). For the web-proxy experiment of Figure 3.1, a workload generator would be implemented by generating per-user-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 existing behaviors.

In contrast, we take an operational approach that models actor behaviors directly through discrete-event simulation. Details for the actor behavior models could come from, for example, empirical traces [56], detailed user behavior profiles [42], real network failure scenarios [23], or an experiment plan in which specific experiment scenarios are described. We name this approach **simulation-based workload generation approach**. Our idea is to give software engineers the ability to quickly and easily express their specific actor behaviors to create a diversity of workloads. We take advantage of the inherent scalability and efficiency of a discrete-event simulation engine to power our approach, enabling us to apply this approach to large numbers of actors exhibiting complicated dynamic behavior. As introduced in Section 2.1.1, discrete-event simulation models a problem through a set of inter-communicating processes. These processes can represent the distributed actors and can naturally model the scheduled actions and dependencies of the four levels we discussed in Section 4.1.

The simulation-based workload generation process is illustrated in Figure 4.2. It allows the engineer to model one or more types of actors as programs written in a common programming language, such as C++, powered by a discrete-event simulation library. Each type represents the actors with the same behavior rules in interacting with the SUE, the testbed, the other actors and itself. With the support of programming, the programmed actors may therefore execute arbitrary functions, maintain arbitrary states, and exchange arbitrary messages. All of their interactions with the SUE and the testbed will be recorded by a special output function provided by a workload output library. This function inserts a virtual timestamp before each interaction, specifying the virtual time when the interaction will happen in an experiment trial. All the interactions within actors (either between different actors or within the same actor) will be processed

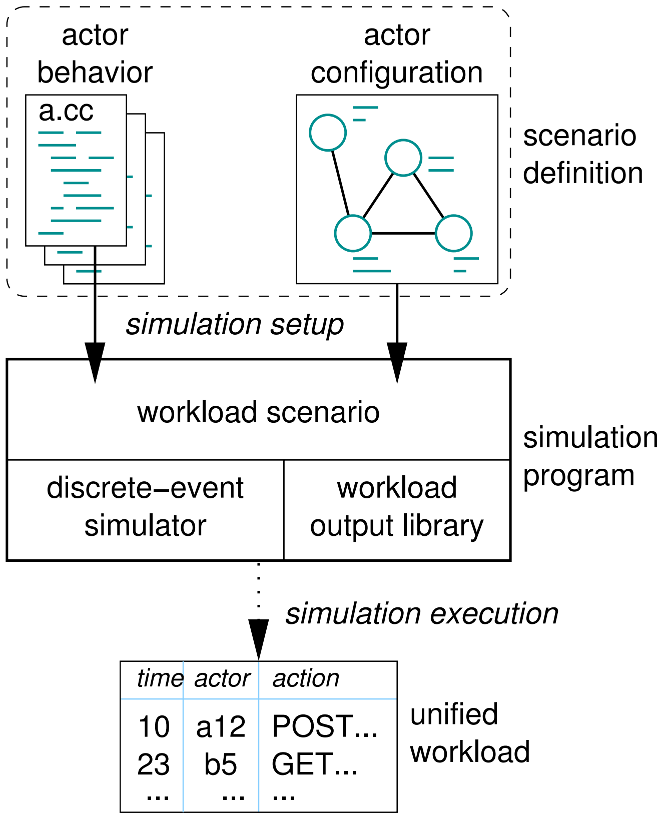


Figure 4.2: Simulation-Based Workload Generation

in the workload generation step through the message exchanging support provided by the discrete-event simulation library.

After programming the actor behavior types, the engineer can populate a workload simulation scenario by specifying a set of workload processes as the instances of those types. These specifications form a **workload configuration**. Based on the workload configuration, each actor will be instantiated to be a simulation process in the simulation library. The implementation of actor behavior types and the actor configurations make up a workload **scenario definition** as shown in Figure 4.3.

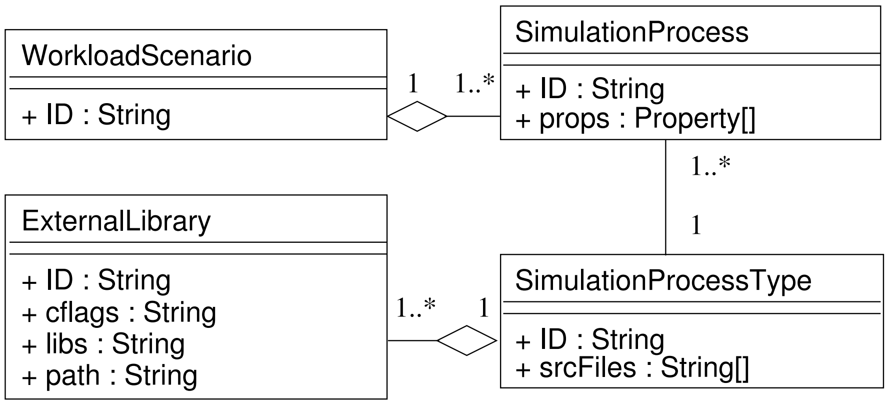


Figure 4.3: Workload Scenario Conceptual Model

Figure 4.3 shows the conceptual model concerning the workload scenario definition. A **WorkloadScenario** with its identifier is defined with a collection of **SimulationProcess** entities. Each **SimulationProcess**, which is an object of a **SimulationProcessType** entity, represents an actor. A **SimulationProcessType** defines an actor behavior type in a list of source files, which are the discrete-event simulation programs we discussed early. These programs may be associated with a collection of **ExternalLibrary** entities that contain external dependencies of the implementation.

Through the **simulation setup**, a workload scenario definition is translated into an executable **simulation program** that aggregates the simulation programs of the actor behavior types and the workload configuration. The simulation program, linked with the discrete-event simulator and the workload output library, executes to produce the desired workload. The workload consists of all simulated interactions between each actor and the SUE or the testbed, which will serve as actions in the experiments with distributed systems.

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

The 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.

With the simulation-based workload generation approach, we are able to generate workloads modeling the actor behaviors of either of the four levels discussed in Section 4.1. For those in level 1 and 3, since there is no inter-actor dependency, the actors can be modeled with a set of independent simulation processes. For example, a simple scenario for level 1 in which each actor has a constant request sequence could be modeled with a simulation process sending out requests to the SUE or the testbed at regular intervals. Instead of being programmed as independent simulation processes, the actors with behaviors of level 2 and 4 are modeled by interdependent simulation processes that can signal other processes or respond to signals from other processes. In the web-proxy example, we generate a workload for a scenario involving level 2 behaviors by simulating two humans browsing two sites and randomly picking URLs that are known to exist. Additionally, each human periodically passes a URL recommendation message to the other, who immediately requests the recommended URL upon receiving the recommendation.

In the next section, we introduce how we implemented the simulation-based workload generation approach in our prototype Weevil. Then we discuss some case studies to further illustrate how the actor behaviors of the four levels get modeled.

## 4.3 Implementation

We have implemented a simulation-based workload generator in our prototype **Weevil** powered by the SSim discrete-event simulation library. [1] The library consists of three classes. The class **Sim** defines the interface to the simulator. It offers the basic primitives for signaling events between processes, and for creating, starting, and stopping processes. SSim library supports two types of simulation processes with two classes, **Process** and **TProcess**. The class **Process** defines reactive processes, while **TProcess** defines sequential processes. A simulation process can be programmed by extending either **Process** or **TProcess**. A reactive simulation process (extending **Process**) is defined by programming the **init**, **stop**, and **process_event** methods. The first two methods define the sequence of behaviors when a process is initialized and stopped. The last **process_event** method defines the functions executed by a reactive process in response to an event signaled from another process or from itself. It is used when the process may have other reactive behaviors in parallel to its routine behaviors. In contrast, each sequential process (extending **TProcess**) can be easily defined by programming the `main` method, which defines the body of the process directly. Each type of processes can use the `self_signal_event` method to signal itself and the `signal_event`

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

method to signal other processes. Besides, a sequential process can also use and the `wait_for_event` method to receive events signaled from other processes.

We created a workload generation library, which extends the SSim discrete-event simulation library. Both the **Process** and the **TProcess** class in the SSim library are wrapped by a class **WeevilProcess**. Thus, any type of actor behaviors is defined in a subclass of **WeevilProcess**. It either implements all the three methods defined in the reactive process class, **Process**, or implements the main body for the sequential **TProcess**. Besides, some methods are added in the library to ease the workload generation use, including the workload output method, `workload_output`. This method outputs each workload line in the following format:

```
event(<timestamp>,<simulation_process_id>,<action>(<parameter>,...))
```

The three arguments for each event, `timestamp`, `simulation_process_id`, and `action` provide the time, the place, and the contents of each action. Each action can be further defined with parameters. The meaning of each parameter is interpreted by the engineer in implementing the actor programs in Chapter 5.

The other part of the scenario definition, workload configuration, is implemented in GNU m4 [1]. We defined a set of GNU m4 declaration macro functions (listed in Appendix A) corresponding to the entities in the workload scenario conceptual model of Figure 4.3. A software engineer thus can configure a workload scenario by calling those declaration macro functions to assign values to the scenario parameters. Our simulation-based workload generator then knows how to setup a simulation program for the workload scenario.

The simulation setup is implemented using m4 macro expansion, through which the workload configuration is translated to a simulation bootstrap program. The bootstrap program first constructs all the configured simulation processes. Each of them is constructed by calling the constructor in the source files of its corresponding **Simu-**

**lationProcessType**s. Then, all the configured process properties are assigned to the process's class variables. At last, the setup program bootstraps the simulation with the **run_simulation** function provided in the SSim library. At this time, all the simulation processes begin to proceed with their behaviors based on the implementation of their corresponding actor behavior types.

## 4.4 Case Studies

In order to demonstrate the flexibility of the simulation-based workload generation approach, we have used our prototype workload generator to produce workloads corresponding to a variety of statistical and operational actor behavior models.

Statistical actor behavior models are not the primary focus of our workload generation approach. Nonetheless, our experience shows that our prototype workload generator, in combination with a scientific library,[2] can easily support these models. The more interesting workloads that the simulation-based approach can produce are those derived from operational behavioral models, and, in particular, those expressing **interdependent** behaviors. The following four case studies illustrate how the simulation-based workload generation approach can model the interdependent actor behaviors of level 2 through 4. There are two case studies for level 2, which study intra-actor dependency and inter-actor dependency, respectivly.

### 4.4.1 Intra-Actor Interdependent Behaviors

One type of level 2 actor behavior is where future requests need to incorporate data from a previous request. As an example, consider a distributed publish/subscribe system where users use an access point to publish notifications and subscribe for notifications of interest. A common behavior for a subscriber could be to periodically subscribe and unsubscribe, with every unsubscription matching the previous subscription. This

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

behavior can be programmed as follows:

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

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

### 4.4.2 Inter-Actor Interdependent Behaviors

Another type of level 2 behavior is 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.[67] 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 simulation 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 simulation process consists of a boolean flag to determine if the represented host is susceptible to probes, the number and time distribution of its probes, and a blacklist 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 in Figure 4.4, including event classes and the simulation process class **Host**. The `Probe` class represents an event signal to the process itself implementing the function of timeout signal. The `Infect` class represents a malicious event signaled to a simulation process in the blacklist. The first half of **Host** implements the parsing and assignment of process properties specified in the workload configuration. In the second half, each actor is defined as a reactive process by implementing the `init` and `process_event` method. The `init` method initializes simulation 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`). Specifically, the scenario is configured using m4 declaration macro functions as shown in Figure 4.5. (The definitions and explanations of the m4 macro functions are listed in Appendix A.) The flexibility of m4 macro operations also provides the software engineer the ability to construct their own macro functions to further save their work in workload configuration. Weevil pre-defines some generally used **functional macro**s as listed in Appendix D, including a loop structure `WVL_SYS_Foreach` and a string operation function `WVL_SYS_Range`. The result of this worm propagation is shown in Figure 4.6.

```
/******Event Classes******/
class Infect : public Event {};
class Probe : public Event {};

/******Reactive Simulation Process Class******/
class Host : public weevil::WeevilProcess {
  //input properties from workload configuration
  bool m_susceptible, m_infected;
  unsigned int m_num_probes, m_probe_interval;
  vector<string> m_blacklist;

 public:
  //parsing and assignment of process properties
  void susceptible(bool b){ m_susceptible = b; }
  void infected(bool b){ m_infected = b; }
  void num_probes(unsigned int n){ m_num_probes = n; }
  void probe_interval(unsigned int n){ m_probe_interval = n; }
  void blacklist(std::string list){
    if ( list.compare("") != 0 ){
      string target;
      int begin = 0;
      int end = 0;
      while (end != string::npos){
        end = list.find_first_of(",", begin);
        target = list.substr(begin, end-begin);
        m_blacklist.push_back(target);
        begin = list.find_first_not_of(" ", end + 1);
      }
    }
  }

  //reactive process implementation
  void init() {
    if (m_infected && (m_num_probes > 0)) {
      m_num_probes--;
      self_signal_event(new Probe(), m_probe_interval);
    }
  }
  void process_event(const Event* e) {
    const Infect * i;
    const Probe * p;
    if ((i = dynamic_cast<const Infect *>(e) != 0){//an Infect event
      if (m_susceptible && !m_infected) {
        m_infected = true;
        workload_output(‘‘INFECT’’);
        init();           // begin infect actions
      }
    }
    else if ((p = dynamic_cast<const Probe *>(e) != 0){//a Probe event
      int fi = rand_normal_range(0, m_blacklist.size()+1);
      signal_event(m_blacklist[fi], new Infect(), 0);
      if (m_num_probes > 0) {
        m_num_probes--;
        self_signal_event(new Probe(), m_probe_interval);
      }
    }
  }
};
```

Figure 4.4: Programmed Worm Behavior

```
define(‘SimulationProcesses’, ‘WVL_SYS_Range(‘U’, 0, 99)’)
WVL_SYS_WorkloadScenario(‘worm’, ‘5000’, ‘SimulationProcesses’)
WVL_SYS_SimulationProcessType(‘MaliciousHost’, ‘host.cc’)
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_SimulationProcess(i, ‘MaliciousHost’)’, SimulationProcesses)
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_SimulationProcessProp(i, ‘susceptible’, ‘1’)’, WVL_SYS_Range(‘U’, 0, 58))
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_SimulationProcessProp(i, ‘susceptible’, ‘0’)’, WVL_SYS_Range(‘U’, 59, 99))
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_SimulationProcessProp(i, ‘infected’, ‘1’)’, WVL_SYS_Range(‘U’, 0, 1))
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_SimulationProcessProp(i, ‘infected’, ‘0’)’, WVL_SYS_Range(‘U’, 2, 99))
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_SimulationProcessProp(i, ‘num_probes’, ‘100’)’, SimulationProcesses)
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_SimulationProcessProp(i, ‘probe_interval’, ‘50’)’, SimulationProcesses)
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_SimulationProcessProp(i, ‘blacklist’, ‘"WVL_SYS_Range(‘U’, 0, 99)"’)’, dnl
        SimulationProcesses)
```

Figure 4.5: Worm Scenario Workload Configuration

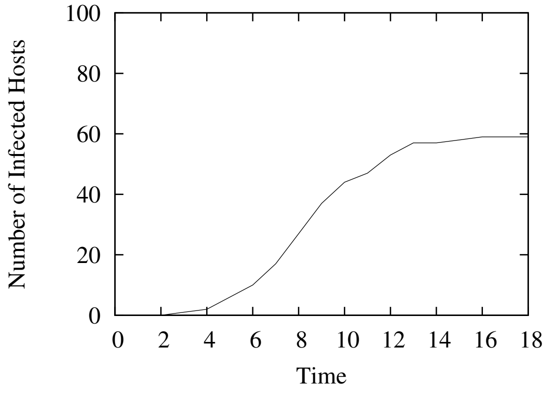


Figure 4.6: Simulated Worm Propagation

### 4.4.3 Level 3 Interdependent Behaviors

In Section 4.1, we described a scenario of the web-proxy example involving actor behaviors of level 3. With the simulation-based workload generation approach, we generate a workload by simulating two humans randomly picking URLs that are known to exist and send out GET requests every n seconds, where the value of n (n=50, 100) may or may not change every 1000 seconds. Modeling such behaviors as a program is straightforward for a software engineer. The following shows the beginning piece of the generated workload.

```
event(100,U0,GET(f80))
event(110,U1,GET(f823))
event(200,U0,GET(f1901))
event(210,U1,GET(f179))
event(300,U0,GET(f296))
event(310,U1,GET(f1382))
event(400,U0,GET(f1240))
event(410,U1,GET(f12))
event(500,U0,GET(f1573))
event(510,U1,GET(f39))
event(600,U0,GET(f895))
event(610,U1,GET(f1034))
event(700,U0,GET(f376))
event(710,U1,GET(f1923))
event(800,U0,GET(f1113))
event(810,U1,GET(f920))
event(900,U0,GET(f340))
event(910,U1,GET(f1))
event(1000,U0,GET(f207,,latency))
event(1010,U1,GET(f934,,latency))
event(1050,U0,GET(f991,latency<2))
event(1060,U1,GET(f12,latency<2))
event(1100,U0,GET(f1970))
event(1110,U1,GET(f132))
event(1150,U0,GET(f1369,latency<2))
event(1160,U1,GET(f932,latency<2))
event(1200,U0,GET(f1500))
event(1210,U1,GET(f230))
...
```

The format of each workload line is `event(<timestamp>, <simulation_process_id>, <action>(<parameter>, ...))`. In this workload, the action is `GET`. The parameters for this action are the **file id**, the **condition** in which action can be performed, and the **trial-execution-time variable** to access during trial execution. Note that the workload lines before timestamp 1000 are all unconditional actions since the first check on request response time happens at 1000 seconds. After timestamp 1000, there is a workload line for every 50 seconds. The lines every 100 seconds are still unconditional workloads since no matter whether n equals to 50 or 100, the actions are issued at least every 100 seconds. But the added workload lines if n equals to 50 are conditional actions depending on the value of the average request response time. For example, the workload line at time `1050` is to issue a request for file `f991` if `latency<2`. The workload lines every 1000 seconds also indicate an extra request to check the value of the average response time. For example, the workload line at time `1000` means to unconditionally issue a request for file `f207` and access the value of the request response time "`latency`".

### 4.4.4 Level 4 Interdependent Behaviors

In Section 4.1, we also described a scenario of the Chord example involving actor behaviors of level 4. With the simulation-based workload generation approach, we generate a workload by simulating a number of humans randomly picking file ids and sending out GET requests every 100 seconds. Besides, the simulation process U1 also accesses the return value to each request he or she issues. If the return value is not 0, which means the request does not get satisfied, U1 sends out a message to U3 asking for help. Since such behaviors cannot be resolved in the workload generation step, we choose to delay the calculation and only include the textual descriptions of the behavior logic in the generated workload:

```
...
event(100,U0,GET(f80,,return_val,return_val!=0,U3))
...
event(200,U0,GET(f1901,,return_val,return_val!=0,U3))
...
event(300,U0,GET(f296,,return_val,return_val!=0,U3))
...
```

In this portion of workload, the action is `GET`, although the `GET` in this example represents the retrieval request of the Chord peer-to-peer system. The parameters for this action are the **file id** to retrieve, the **condition** in which the action can be performed, the **trial-execution-time variable** to access during trial execution, the **message condition** in which the associated actor needs to send messages to another actor, and the **message target** to send the message to. For example, the workload line at time `100` is to unconditionally issue a request for file `f80`, and if the request return value is not 0, a message for help is sent to `U3`.

# Chapter 5

# Configuration Modeling

Upon readiness of the workloads, our trial automation framework applies them in an experiment trial. The actions contained in the workloads are translated into real requests, which in turn stimulate an SUE execution in a realistic network environment for the purpose of collecting valuable experiment results. These activities are automated in the next two trial steps covered by our framework, the trial deployment and execution step, as well as the trial post-processing step.

To achieve a high degree of automation and to build a generic, programmable automation environment, we have taken a model-based generative approach to the design of our trial automation framework. In this framework, generative techniques, as discussed in Section 2.1.2, are used to transform high-level trial configuration directives into the implementation of a trial control system. This trial control system then automates trial deployment and execution, as well as trial post-processing.

Our model-based generative approach provides three main advantages over manual approaches: (1) engineers are relieved of the burden of creating and maintaining a large volume of trial control scripts, and instead must only deal directly with a relatively concise set of configuration parameters; (2) models can be shared among experiment trials and easily tweaked between trials; and (3) The generative capabilities transparently handle much of the complexity brought about by the scale and heterogeneity of the distributed systems and their execution environments. Generallyspeaking, experimenting with distributed systems becomes easier and more cost-efficient with the model-based generative approach.

Figure 1.3 provides overview of the automated trial process based on the model-based generative approach. Figure 5.1 provides its amplified version. In this figure, 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 the input model configurations provided by the engineer. White ovals represent generated control scripts, programs and auxiliary files. The cross-hatched ovals represent data finally collected by the software engineer as the trial results. Solid arrows represent normal input/output data flow, whereas dotted arrows represent the execution of scripts. As seen in Figure 5.1, the trial deployment and execution step and the post-processing step are divided into five main phases, configuration modeling, trial setup, trial deployment, trial execution, and post-processing. The configuration modeling phase is the only phase requiring an engineer's interaction to provide trial inputs, including a user workload, an optional environment workload, and a trial configuration file. All of the other phases get inputs from their previous phases, perform actions, and output intermediate or final results. In specific, the workload(s) and the configuration file are translated into a customized trial control system, which is made up of a set of trial scripts, programs, and auxiliary files, adaptively through the **trial setup** phase. Then in the **trial deployment** phase, the trial control system is deployed onto the testbed. The deployed control system starts up the components of the SUE and synchronizes the actors to get ready for the **trial execution**. Upon the completion of the trial deployment, the control system decides a time to start up all the user actors and the environment actors at the same time. These actors process their workloads with the actor programs to drive the execution of the SUE. The events and states generated during the execution phase are recorded in log files. Finally, in the **post-processing** phase, all the diagnostic data derived from the log files are collected back to the software engineer with the testbed

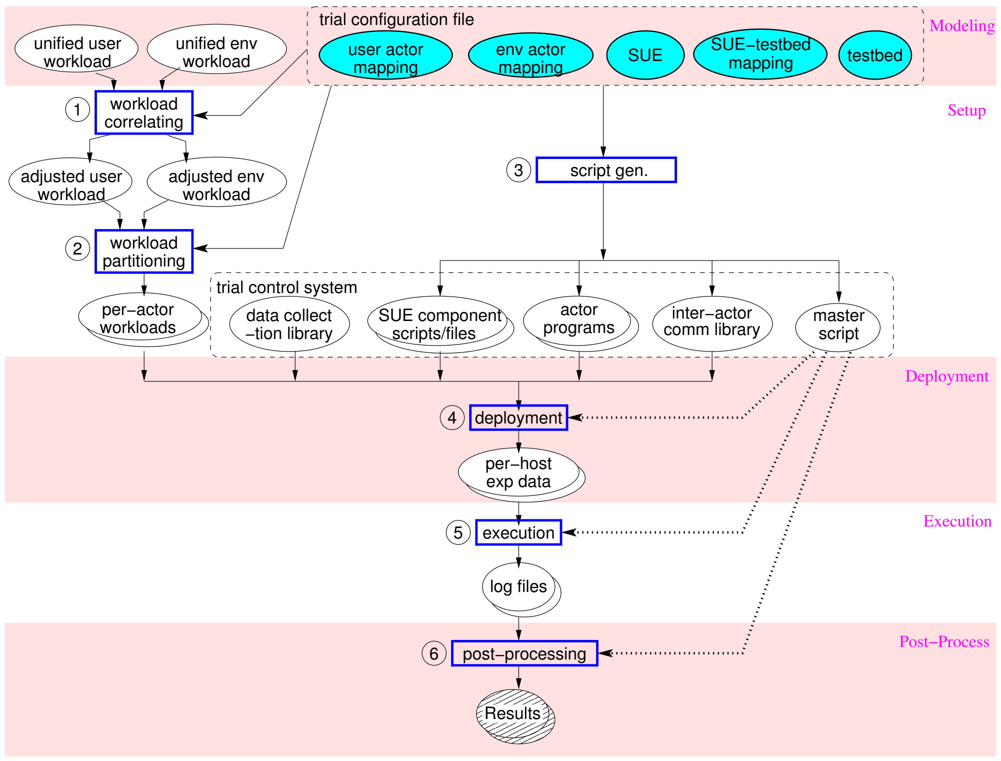


Figure 5.1: Automated Experiment Trial Process

cleaned up.

In this chapter, we discuss the configuration modeling phase, which is the base of the automation in the following phases. In this phase, a software engineer provides complete descriptions of an experiment trial. Although the architectures and algorithms of distributed systems vary greatly from system to system, they share the commonalities in that they all are made up of distributed communicating components and all of their clients access the service through the system client APIs. Similarly, an existing network environment can be specified by declaring its included hosts. These commonalities illustrate a common way for us to specify different experiment trials with different distributed systems by configuring a suite of configuration models.

As we have discussed in Chapter 1, a particular experiment trial is related to four primary concepts: the **SUE**, the **testbed**, the **user actor**, and the **environment actor**. We thus describe an experiment trial from these four main aspects. The SUE and the testbed are both described with two primary configuration models, the **SUE model** and the **testbed model**. All the pre-computable behaviors of the actors are already described in the workloads discussed in Chapter 4, the **user workload** and the **environment workload**. By now, these four aspects are still independent. However, an experiment trial is an aggregation of these four aspects. Thus, three additional mapping models associating the four aspects are also included in the model space, including the **SUE-to-testbed mapping**, the **user actor mapping**, and the **environment actor mapping**. The last two mapping models associate each distributed workload process with an experiment entity through the implementation of the actor programs, in which all the actor behaviors incomputable in the workload generation step are described. These configuration models are represented by the dark ovals along the top of Figure 5.1. We discuss them one by one in the following sections.

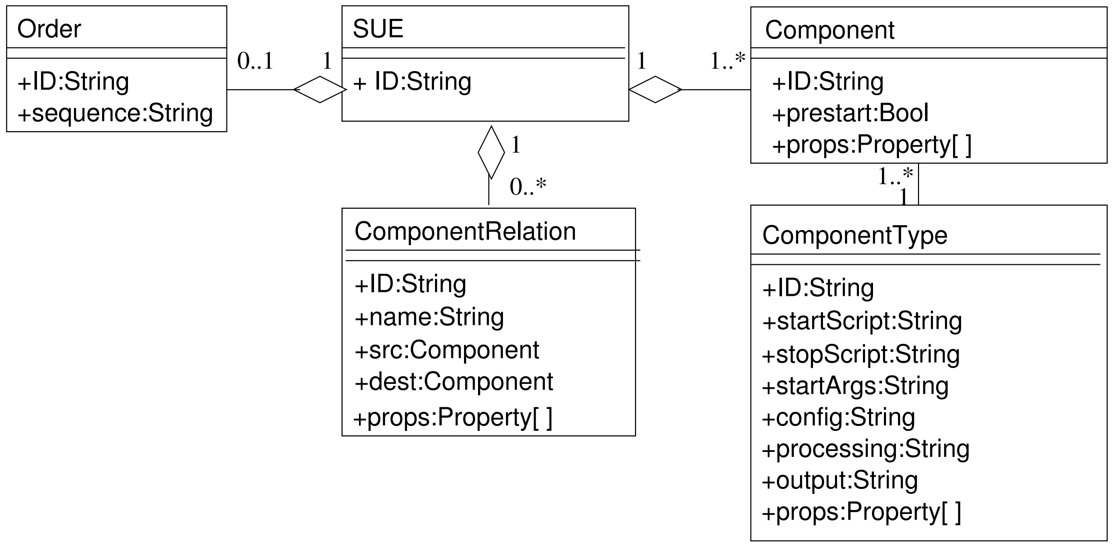


Figure 5.2: SUE Configuration Model

## 5.1 SUE Model

The SUE model, as shown in Figure 5.2, specifies the SUE components, its organization, its execution order, and the primary commands acceptable by the SUE. An SUE is made up of a collection of typed **Component**s. There can be more than one type of components in an experiment trial, such as the web-proxy example of Figure 3.1, which includes three types of components, the browsers, the proxies, and the servers. Each component type is defined in the **ComponentType** entity. Each component is declared as an instance of a **ComponentType**. Such kinds of OO-like design allows common attributes to be shared among all components of the same type. If in an experiment, an engineer conducts a set of trials to study a system with different numbers of components without introducing any new types of components, he or she would only need to declare different numbers of component instances in each trial without the need to modify the **ComponentType** entity.

The **ComponentType** entity defines the execution primitives, i.e., how to start, stop, and post-process the components of this type. As shown in Figure 5.2, the **startScript** attribute, which is necessary for the **ComponentType** entity, contains the template of the script to start up a component of this type. The script may need some parameters to configure the SUE when it is started up. Thus, the optional attributes, **startArgs** and **config**, contains the templates of parameters passing to the **startScript**. The attribute **startArgs** provides the template of the command-line parameters to the start script when the trial is started. The **config** attribute is necessary if the software of this component type configures its parameters in a configuration file instead of through command-line parameters. These two attributes only provide the parameter values when an experiment trial is started. Their values may change during trial execution if an environment actor brings down a component and re-starts it using different command-line parameter values or a different configuration file. The **stop-**

**Script** attribute contains the template of the stop script. The attributes **processing** and **output** are used to specify the post-processing rules for the log files of a component type. These rules will be translated into post-processing scripts for each component. We delay our discussion on them until Section 6.4 where the trial post-processing step is explained.

Although an SUE is defined to be made up of a collection of components, these components are not required to stay in the trial execution all the time. In some experiments like robustness testing, the SUE components change during the trail execution phase. For example, to study the robustness of a system in unreliable network environments, a software engineer may configure an experiment trial in which some system components fail and join from time to time. The **prestart** attribute for each component specifies if the component is started before the trial begins to execute or it may join later during the trial execution phase through an environment actor, which we will discuss in Section 5.3.3.

The **ComponentRelation**s contained in the SUE model are used to represent any binary associations between components. These are optional and entirely system specific. For instance, in Figure 3.1 there are three relations (shown as dashed lines): component P1 is a “proxy” for B1, P2 is a “proxy” for B2, and P1 is a “peer” of P2. In general, relations are used in situations where one component references properties of another component in its start script or configuration file to bootstrap.

**Order** entities are used to represent the necessary or preferred order in which the components should be started. This is also optional and entirely system specific, since some distributed systems require certain components to be ready before others. For example, Siena requires a parent server to be ready before its children can start, but the children can start in any order. Similarly, a bootstrap Chord instance needs to be available before any other nodes can join a Chord network. However, the order to start up components does not matter for some distributed systems. For instance, in the web-proxy example, the two proxies work cooperatively and access web pages from the two servers. If we use **Squid** [27] as the web proxies in this example, since Squid software does not require the sibling proxies to start in any specific order, the **order** entity will not be necessary.

## 5.2 Testbed Model

Testbed is the network environment to which an experiment trial is deployed. Our vision is to support experiments with distributed systems in realistic network environments. As introduced in Section 2.1.3, the actively used distributed testbeds can be sorted into three categories, local-area networks, wide-area overlay networks, and emulated networks. Our trial automation framework supports these different types of testbeds by making minimal assumptions about the testbed. It only requires an account on each testbed machine accessible through user-level remote shell access. The testbed network property configuration is not covered in our framework. Instead, we use the testbeds already set up with the testbed mechanisms. Such mechanisms have been well supported in the public testbeds like PlanetLab and Emulab. As shown in Figure 5.3, a **Testbed** is defined as a collection of **Host**s.

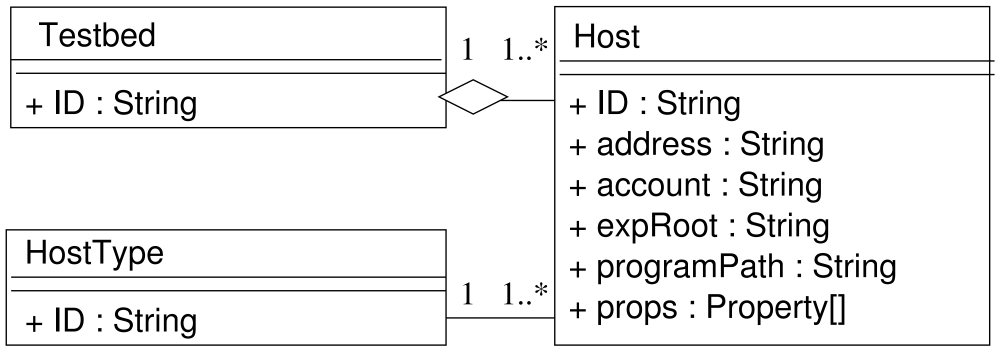


Figure 5.3: Testbed Configuration Model

The **Host** entity provides the network identity and resources provided by a testbed host. Each testbed host is an **account** on a network **address**. For a testbed on the PlanetLab, the account on each testbed host is actually the PlanetLab slice name to which the engineer is assigned. The **expRoot** attribute specifies the directory allocated to the experiment trial on each host. All the trial-related files are under this directory. The **programPath** specifies the path to the program that is necessary for the trial automation system to execute the specific trial. For example, if the generated trial scripts in the trial automation system are Bourne Shell scripts, the **programPath** should provide the local path to the program `sh` on each host.

To support the deployment of the SUE on heterogeneous testbeds, the testbed model also includes a **HostType** entity that is used to partition the testbed hosts into categories needed for specifying which binary packages of the SUE should be deployed on a host type. We will discuss this later in Section 5.3.1.

## 5.3 Mappings

The two models described above and the workloads are designed to be largely independent of each other. This gives the engineer a fair amount of flexibility in composing experiment trials. The composition is implemented through three mappings among the four aspects.

### 5.3.1 System-to-Testbed Mapping

For a specific experiment trial, the system components are deployed onto the testbed hosts for trial execution. The way in which the system components are deployed is specified by configuring the system-to-testbed mapping model. It associates the SUE with the testbed by specifying on which testbed host each system component should reside. A single host can serve multiple components. The other part of the system-to-testbed mapping model is to associate a **ComponentType** with a **HostType**. The source code for a component type may compile to different binary distributions on different platforms. For example, the Chord source code compiles to different binary distributions on **Linux** machines and on **Solaris** machines. Thus, the other part of the SUE-to-testbed mapping specifies which binary distribution to use for a component type (**ComponentType**) if they are deployed onto the hosts of a specific type (**HostType**).

### 5.3.2 User-Actor Mapping

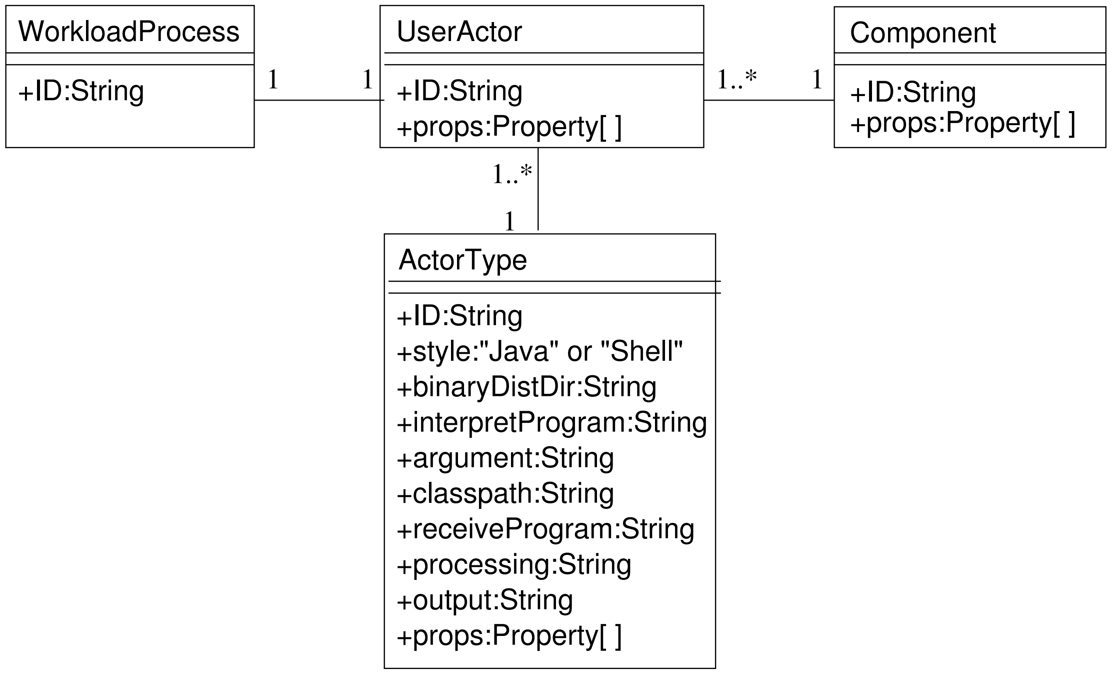


Figure 5.4: User-Actor Mapping Configuration Model

The user workload represents a list of system service calls sent from distributed system users to different system access points of a distributed system. To apply the user workload in trial execution, the second mapping model, as shown in Figure 5.4, maps the user workload to the SUE through user actors. In specific, each workload process (i.e., the simulation process in our simulation-based workload generation approach) in the workload is mapped to a single SUE component through a **UserActor** that is declared as an instance of an **ActorType**.

The **ActorType** entity represents a type of actor whose actor programs share the same templates. As mentioned in Section 4.1, actor behaviors are modeled with the combination of workloads and actor programs. Actor programs are used to solve the variables included in the workload that cannot be pre-computed in the workload generation step. In each row of Table 4.1, we list the methods used to model the four levels of actor behaviors. Two of them, the trial-execution-time information analysis and the trial-execution-time inter-actor communication, are implemented in actor programs.

We design the actor programs to include two parts, the interpretation program **interpretProgram** and the message receiving program **receiveProgram**. The interpretation program implements a sequential process. It implements the sequence of actor behaviors according to the textual workload. In contrast, the message receiving program implements a reactive process to react to the messages from other actors. It implements the interpretation of the inter-actor trial-execution-time information and the possible reactive actor behaviors. We discuss these two types of programs in detail.

**Interpretation Program**

The interpretation program understands how to stimulate the trial execution as dictated by the workload. For each actor, it sequentially interprets each workload line belonging to the workload process associated to the actor. In specific, in the interpretation program for a user actor, it translates each user workload line to one or more system service calls. Since each user actor models a system user's behaviors, it only accesses the SUE through the system APIs and possibly interacts with other user actors. In other words, it cannot makes changes to the testbed, nor will it try to access the testbed information. Thus, all the conditions contained in the conditional user workload lines can be resolved by analyzing the trial-execution-time information returned by the SUE to the system service calls. A user actor may also communicate with other user actors during trial execution exchanging their own workload processing status.

The implementation of an interpretation program is supported by a workload interpretation library. This library controls the time to stimulate the SUE through its implementation of the cycle of reading the workload and translating each virtual timestamp in the workload to a real time in trial execution. It calls the user-provided interpretation program upon every workload line. Thus, in the workload interpretation program, the software engineer is expected to simply implement the function of parsing the different actions in the workload and issuing its corresponding real requests. The possible requests issued by user actors can be sorted into two categories:

- System service calls using the system APIs: The function of actors is to stimulate the trial execution. The user actors implement this function with system service calls through system APIs. In Figure 3.1, the two different user actors use the two APIs, `wget` and `galeon`, to turn the "GET" actions for the URLs contained in the workload into real HTTP requests. The user actors can also catch the system responses to the service calls in helping decide how to interpret the following workload lines.
- Inter-user-actor communication commands: As discussed in Section 4.1, a user actor's behaviors of the level 4 may depend on the trial-execution-time information collected by another user actor. This information cannot be pre-determined without trial execution and trial-execution-time communication between user actors. The inter-user-actor communication commands are used to transfer such trial-execution-time information between user actors.

The interpretation program provided by a software engineer may use other entity attributes in the trial configuration model. For example, the **expRoot** attribute can be included in the program to locate the trial scripts on different testbed hosts. These entity attributes can be included in the attribute **argument** as the arguments passing to the interpretation program or included in the interpretation program directly. They will then be instantiated to the different values to generate the specific interpretation programs customized to different user actors during the trial setup phase.

**Message Receiving Program**

The message receiving program, as a part of the trial-execution-time inter-actor communication mechanism, is in charge of receiving and interpreting messages from other actors. For each actor, it reactively interprets the inbound communication message and possibly issues real requests reactive to the message.

Similar to the interpretation program, the message receiving program provided by a software engineer may also use other entity attributes in the trial configuration model. These entity attributes are included in the message receiving program directly. As discussed in Section 6.1, they will be expanded to the values of the attributes to generate the specific message receiving programs customized to different user actors during the trial setup phase.

Besides the two primary programs, the actors sometimes are also in need of some other files and programs to aid their execution. For example, if an actor's behavior is to store files into a Chord network, all those files need to be ready for the actors. The attribute **binaryDistDir** thus lists all the necessary directories, the files under which will be deployed during trial deployment.

Similar to those of the **ComponentType** entity, the attributes **processing** and **output** are used to specify the post-processing rules for the log files of the actor types: the processed trial results are copied back from each actor's working directory to the trial master.

### 5.3.3 Environment-Actor Mapping

The environment workload represents a list of distributed changes to the system execution environment. To apply the environment workload in trial execution, the last mapping model maps the environment workload to the trial execution environment through environment actors. In specific, each workload process (i.e., the simulation process in our simulation-based workload generation approach) in the environment workload is mapped to a single testbed host or an SUE component through an

**EnvironmentActor** that is declared as an instance of an **ActorType**.

Different from a user actor, which represents a real system user, an environment actor does not necessarily have a real corresponding entity. It could represent a real malicious intruder who constantly injects garbage packages to the network to affect the system execution. It could also capture a phenomenon that a host always fails whenever its load is too high. All the environment actors as a whole model the environment variation scenarios. Such scenarios can be implemented using the same mechanism as the operation of the distributed systems. Similar to a user actor, an environment actor is distributed on the testbed performing actions from time to time, at the same time also communicates with other environment actors. All the distributed environment actors cooperate to achieve an aggregated scenario described in the environment workload.

In the trial configuration modeling, each environment actor is also declared as an instance of an **ActorType** discussed in Section 5.3.2. The **ActorType** of environment actors are defined with the same entity attributes as those of user actors, while the actor programs may issue different categories of requests. This is because, the environment actors can take many routes in order to learn about and make changes to the environment, not necessarily through the SUE APIs. Instead of communicating with user actors, they communicate with other environment actors to exchange environment status. In specific, the possible requests issued by environment actors can be sorted into the following categories:

- System service calls using the system APIs: An environment user could also use the APIs provided by the SUE. However, its purpose is not to operate the SUE, but to access information from the SUE to guide its behaviors of changing the environment or to make changes to the SUE components. For example, an environment actor in the Chord network experiment could call its associated Chord node's stop script to terminate the Chord node. Another environment actor could call the client program to get the routing information of the Chord network.

- Testbed commands using the testbed APIs: A direct way to change the environment is to modify the testbed through the testbed APIs, including shell commands and testbed-specific APIs. Suppose the Chord network example is conducted on an Emulab testbed, Emulab provides the ability to inject events into the testbed dynamically via the **Testbed Event Client (tevc)**. For example, the testbed-specific command "`tevc -e testbed/exampleexp now link0 down`" breaks **link0** in the testbed when the command is issued.

- Other software commands: In addition to the SUE and the testbed, other software, such as a fault injector, may be installed to affect the trial execution environment. The environment actors thus could send out commands using the APIs of other software.

- Inter-environment-actor communication commands: Some environment variation scenarios are complicated enough to involve the level 4 actor behaviors. For example, in a scenario, the host that has the heaviest load among all the testbed hosts will be turned down. In the distributed environment, the load on each host has to be collected through a set of environment actors each of which is distributed on a testbed host. To find out the maximum load value among them, some distributed protocol needs to be applied to these actors. The protocol requires the inter-environment-actor communication.

## 5.4 Implementation

We have implemented the model-based generative approach in our prototype Weevil, in which all the above trial configuration models are implemented in GNU m4. In specific, we defined a **declaration macro function** for each entity of the models. The attributes in each entity are the parameters to the function. The configuration modeling is achieved by calling those declaration macro functions to instantiate the models. In other words, the declaration macros will define a set of **property macro**s serving as properties of an experiment trial. All these property macros can be used as parameters in other declaration macros. All the declaration macro functions and the property macros defined by them are listed in Appendix B. Other property macros defined by the combination of declaration macros are listed in Appendix C. We discuss some declaration macro functions here to illustrate our configuration modeling implementation and to help illustrate the case studies and meaningful experiments in Chapter 8.

The top-level declaration macro function defines the overview of an experiment trial, including all of its necessary entities and properties. The following is the macro's signature and an example showing how it gets called to declare the Chord example in Figure 3.2.

```
WVL_SYS_Trial(<ID>, <userworkload>, <useractors>, <environmentworkload>, <environmentactors>,
  <testbed>, <sue>, <componentHosts>, <parallel>, <timeout>, <clean>)
```

Example:

```
WVL_SYS_Trial(‘Exp_0’, ‘Wkld_0’, ‘WVL_SYS_Range(‘A’, 0, 5)’, ‘Env_Wkld_0’, ‘WVL_SYS_Range(‘B’, 0, 3),
  ‘Bed_0’, ‘ChordRing’, ‘WVL_SYS_Range(‘CHMap’, 0, 5)’, ‘30’, ‘2000’, ‘Y’)
```

As shown in the parameters to this macro function, the four main entities are the SUE, the testbed, the user workload, and optionally the environment workload. The three mapping entities between them are the component-to-host mapping, the user-actor mapping, and optionally the environment-actor-mapping. Each optional parameter is assigned to a null string if it is unnecessary for the experiment trial. All these entities are briefly referred in this function by their identities. Each entity will be configured further with other declaration macro functions. Note that the functional macro **WVL_SYS_Range** included in the example is used to construct a list of identities sharing the same prefix.

The last three parameters to this macro function define the "breadth" and the "length" of a trial. The "breadth" is specified by **parallel**. Since Weevil supports parallel communication with each testbed host to be discussed later in Section 6.5, "breadth" defines the supported maximum number of parallel processes to be controlled at the same time. The "length" is specified by **timeout**, after which the trial has to be forced to terminate. The last parameter **clean** specifies if the workspace of the trial (i.e., the **expRoot** attribute that we discussed in Section 5.2) should be cleaned up after the trial terminates.

Serving as our way to apply a workload (either a user workload or an environment workload) to an experiment trial, another declaration macro function further configures the workload entities appearing in the above macro function.

```
WVL_SYS_Workload(<ID>, <filename>, <processes>)
```

Example:

```
WVL_SYS_Workload(‘Wkld_0’, ‘user_0.wkld’, ‘WVL_SYS_Range(‘U’, 0, 5)’)
WVL_SYS_Workload(‘Env_Wkld_0’, ‘env_0.wkld’, ‘WVL_SYS_Range(‘I’, 0, 3)’)
```

This declaration macro function associates an abstract identity to the workload. It also points out the workload processes associated to the workload lines, which will be used in the mapping configurations.

As seen in Figure 5.3, a distributed testbed is modeled as a collection of distributed hosts. We implemented the testbed model through a set of corresponding declaration macro functions:

```
WVL_SYS_Testbed(<ID>, <hosts>)
WVL_SYS_HostType(<ID>)
WVL_SYS_Host(<ID>, <address>, <account>, <weevilRoot>, <bourneShell>, <java>, <type>)
WVL_SYS_HostProp(<hostID>, <propertyName>, <propertyValue>)
```

Example:

```
define(‘hrange’, ‘WVL_SYS_Range(‘H’, 0, 4)’)
WVL_SYS_Testbed(‘Bed_0’, ‘hrange’)
WVL_SYS_HostType(‘Linux’)
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_Host(‘H’i, ‘node’i‘.example.Weevil.emulab.net’, ‘ywang’, ‘/tmp/weevil’,
  ‘/bin/sh’, ‘/usr/java/j2sdk1.4.2_07/bin/java’, ‘Linux’)’, hrange)
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_HostProp( ‘H’i, ‘alias’, ‘node’i)’, hrange)
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_HostProp( ‘H’i, ‘ip’, ‘10.1.1.’eval(i+2))’, hrange)
```

Weevil constructs each specific trial control system mainly in Bourne shell scripts. Thus, each **Host** has the attribute **bourneShell** to provide its local path to the program **sh**. The **java** attribute is also necessary since Weevil provides a **Java** library to support the implementation of actor programs in Java. The example configures a testbed on the Emulab. As the example shows, it does not require any special care other than that required for a local network testbed. We will discuss further on the versatility of the testbed modeling through case studies in Section 8.2.1.

Similarly, we further configure an SUE model through a set of declaration macro functions corresponding to Figure 5.2:

```
WVL_SYS_SUE(<ID>, <components>, <relations>, <order>)
WVL_SYS_ComponentType(<ID>, <startScript>, <stopScript>, <startArgs>, <config>, <processing>, <logs>)
WVL_SYS_Component(<ID>, <Type>, <preStart>)
WVL_SYS_ComponentRelation(<ID>, <name>, <src>, <dest>)
WVL_SYS_ComponentOrder(<ID>, <sequence>)
WVL_SYS_ComponentTypeProp(<componentTypeID>, <propertyName>, <propertyValue>)
WVL_SYS_ComponentProp(<componentID>, <propertyName>, <propertyValue>)
WVL_SYS_ComponentRelationProp(<componentRelationID>, <propertyName>, <propertyValue>)
```

Example:

```
dnl
dnl ----------------------- SUE -----------------------------
dnl
define(‘CRange’, ‘WVL_SYS_Range(‘C’, 0, 5)’)dnl
WVL_SYS_SUE(‘ChordRing’, ‘CRange’, ‘WVL_SYS_Range(‘COC’, 1, 5)’, ‘Order_0’)
dnl
dnl ----------------------- ComponentType -----------------------------
dnl
WVL_SYS_ComponentType(‘ChordNode’, ‘Chord_startScript’, ‘Chord_stopScript’, ‘Chord_args’,,, dnl
‘WVL_Trial_Name‘_’WVL_Component_ID‘.log’’)dnl
dnl
dnl -----Chord Start Script-----
define(‘Chord_startScript’, dnl
‘theSoftwarePath/adbd -d theCompPath/db_file -S theCompPath/db &
echo $! >theCompPath/WVL_Trial_Name‘_’WVL_Component_ID‘_1.pid’
sleep 2
theSoftwarePath/lsd -j $1:$2 -p ‘WVL_Component_’WVL_Component_ID‘_ListenPort’
-d theCompPath/db -S theCompPath/sock -C theCompPath/ctlsock
-l ‘WVL_Host_’theHost‘_inner_ip’ -L theCompPath/WVL_Trial_Name‘_’WVL_Component_ID.log
-T theCompPath/trace &’)dnl
dnl
dnl -----Chord Start Arguments-----
define(‘Chord_args’, ‘dnl
define(‘bs_comp’, ‘WVL_ComponentRelation_’WVL_Component_ID‘_bootstrap’)dnl
define(‘bs_host’, ‘WVL_ComponentHost_’WVL_SYS_Echo(bs_comp)‘_host’)dnl
define(‘bs_address’, ‘WVL_Host_’WVL_SYS_Echo(bs_host)‘_alias’)dnl
bs_address ‘WVL_Component_’bs_comp‘_ListenPort’’)dnl
dnl
dnl -----Chord Stop Script-----
define(‘Chord_stopScript’, dnl
‘read adbdpid <theCompPath/WVL_Trial_Name‘_’WVL_Component_ID‘_1.pid’
kill $adbdpid
read pid <theCompPath/WVL_Trial_Name‘_’WVL_Component_ID.pid
echo $pid
kill $pid’)dnl
dnl
dnl -----------------------Component----------------------------
dnl
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_Component(i, ‘ChordNode’, ‘Y’)’, CRange)dnl
dnl
dnl -----------------------ComponentProp----------------------------
dnl
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_ComponentProp(i, ‘ListenPort’, eval(27882+(i/50*2)))’, CRange)dnl
dnl
dnl -----------------------ComponentRelation----------------------------
dnl
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_ComponentRelation(‘C0’i, ‘bootstrap’, i, ‘C0’)’, CRange)dnl
dnl
dnl -----------------------ComponentOrder----------------------------
dnl
define(‘last1’, ‘N99, 1’)dnl
WVL_SYS_ComponentOrder(‘Order_0’, ‘WVL_SYS_Foreach(‘i’, ‘‘N’i, 1, ’, WVL_SYS_Range(‘’, 0, 98))last1’)dnl
```

The first five functions correspond to the five entities in the SUE model. Other than the reserved entity attibutes, Weevil also allows some specific attributes, which cannot be generalized in the trial configuration model, to be assigned to some entities, such as the last three ones in this set of SUE declaration macro functions. The SUE portion of the example configuration becomes a little complicated, especially the configuration of the **ComponentType** entity, which includes not only entity identities and their relationships, but also scripts and strings containing the property macros that was defined by other declaration function calls. A reserved macro name `WVL_Component_ID` refers to each specific SUE component of this type. Thus, the macro `‘WVL_ComponentRelation_’WVL_Component_ID‘_bootstrap’` will expand to the bootstrap node of an SUE component. Its expanded value comes from the declaration macro function `WVL_SYS_ComponentRelation`. In the above example, this macro will always expand to **C0**, the common bootstrap node. Yet another two property macros that are defined by a combination of other declaration macro functions, **theSoftwarePath** and **theCompPath** will be expanded to the real local paths on a testbed host. The former path refers to the directory where the SUE binary distribution is deployed, while the latter one points to the directory where each specific SUE component keeps its trial scripts and execution data. Both of them learn how to expand through the **expRoot** configuration of the **testbed** entity and the Weevil workspace directory structure as shown in Figure 6.4. Through these property macros, the above **startScript** configuration is the template for the following two commands to run each Chord node[1] :

```
adbd -d db -S sock

lsd -j hostname:port -p port
[-d <dbsocket or dbprefix>]
[-v <number of vnodes>]
[-S <sock>]
[-C <ctlsock>]
[-l <locally bound IP>]
[-m [chord|debruijn]]
[-b <debruijn logbase>]
[-s <server select mode>]
[-L <warn/fatal/panic output file name>]
[-T <trace file name (aka new log)>]
[-O <config file>]
```

Note that the $1 and $2 in the **startScript** are the two parameters to the component start script, which are configured in the **startArgs** attribute. They are not contained in the **startScript** since their values may change during the trial execution if the initial bootstrap node **C0** gets turned down.

Another set of the declaration macro functions corresponds to the two mapping models related to actors, the user-actor mapping and the environment-actor mapping.

```
WVL_SYS_ActorType(<ID>, <style>, <binaryDistDir>, <interpretProgram>, <argument>, <classpath>,
 <receiveProgram>, <processsing>, <logs>)
WVL_SYS_Actor(<ID>, <simulationProcess>, <type>, <entity>, <dir>)
WVL_SYS_ActorProp(<actorID>, <propertyName>, <propertyValue>)
```

Example:

```
WVL_SYS_ActorType(‘ChordUser’, ‘Shell’, ‘./actor’, ‘chordrequester.sh’, , , , ,
 ‘WVL_Trial_Name‘_’WVL_Actor_ID‘.log’’)dnl
WVL_SYS_Actor(‘A0’, ‘U0’, ‘ChordUser’, ‘C0’, ‘./files/U0’)
...
WVL_SYS_ActorType(‘FaultInjector’, ‘Shell’, ‘./actor’, ‘faultinjector.sh’, , ,
 ‘messagereceiver.sh’, , ‘WVL_Trial_Name‘_’WVL_Actor_ID‘.log’’)dnl
```

[1] http://pdos.csail.mit.edu/chord/howto.html

```
...
WVL_SYS_Actor('B2', 'I2', 'FaultInjector', 'H3', )
...
```

Weevil provides a library for actor program implementation, including a wrapper for user-provided workload interpretation programs implemented in shell script or Java and the communication primitives for trial-execution-time inter-actor communication. For example, the two shell scripts used to send a file or a message to other actors can be called in the actor programs in this way:

```
sh weevil-sendFile.sh <a file listing the identities of the target actors> <file to transfer>

sh weevil-sendMessage.sh <a file listing the identities of the target actors> <message to transfer>
```

The shell script **weevil-sendFile.sh** transfers a file to all the actors listed in its first parameter and saves the file under each **theActorPath**. The other script **weevil-sendMessage.sh** securely connects to each remote host that each target actor is located on and executes the receiving program of each of the target actors with the transferred message as the program parameter. For example, if `B2` is listed as the target actor, its receiving program **messagereceiver.sh** will be executed. These two shell scripts are both implemented using the vxargs parallel execution tool[2] so that each actor can communicate with other actors fast enough to keep the communication as non-intrusive as possible.

In the workload interpretation program, the software engineer is expected to simply parse the action included in each workload line and send out real requests. For example, the following conditional statement in **chordrequester.sh** translates the action `GET(key)` in a user actor's workload to the real requests through a Chord client program.

```
...
if [ $command == "GET" ]
then
  sh theTimeStampScript theActorPath/WVL_Trial_Name'_'WVL_Actor_ID.log "get $parameter"
  time -f %e -o theActorPath/WVL_Trial_Name'_'WVL_Actor_ID.log -a theSoftwarePath/filestore
    theCompPath/sock -f $parameter 1 2>>theActorPath/WVL_Trial_Name'_'WVL_Actor_ID.log &
fi
...
```

[2] http://dharma.cis.upenn.edu/planetlab/vxargs/

There are two requests to implement the `GET` action. The first is to call the shell script `theTimeStampScript` provided by Weevil data collection library, which prefixes a trial-execution-time timestamp to a string and saves the prefixed string in a log file. The timestamp stamps the global time synchronized for all the testbed hosts. The second request calls the program `filestore` and saves the request latency and possible error messages in the actor's log file.

# Chapter 6

# Experiment Automation

As mentioned in Chapter 1, the primary contribution of this thesis is to use the generative approach to provide a higher level service than other available work. As introduced in Section 2.2, the available approaches and tools for automating experimentation either focus only on centrally controlling user-provided scripts, or are applicable to only one type of distributed architecture, or instead aim at deploying and monitoring distributed services. They cannot meet the increasing needs of distributed system developers to quickly evaluate their systems. The normal practice of a software engineer in experimenting with a distributed system is to manually program experiment control scripts. These scripts implement the function of deploying the SUE, starting up the SUE, and injecting requests to the SUE. In this thesis, instead of the normal practice of manually programming the scripts, a trial control system is generated from the set of trial configuration directives described in Chapter 5. The generated control system that is specific to the configured trial is used to achieve the trial automation in a more flexible way than other available work. As shown in Figure 5.1, upon the completion of the configuration modeling phase, the following four phases: trial setup, deployment, execution and post-processing, can be fully automated.

## 6.1 Trial Setup

The goal of the setup phase is to construct a trial control system based on the workload(s) from Chapter 4 and the trial configuration directives from Chapter 5. The constructed trial control system is customized to the configured trial. It is made up of a set of customized control scripts, programs, libraries, and per-actor workloads. The setup phase is achieved through three main actions, workload correlating, workload partitioning and script generation.

**Workload correlating** (action 1 in Figure 5.1), which is optional in the trial setup phase, is the action to correlate the environment workload with the user workload. The two input workloads are independent from each other. Although the user workload can capture the interdependency of the user behaviors and the environment workload can capture the environmental changes as discussed in Chapter 4, they are independent from each other. But in reality, the distributed system execution scenarios are very complicated. A sequence of operations on a distributed system may only cause exceptions or show interesting phenomena in a specific environment. To help model such scenarios, based on the actor mapping configuration, the workload correlating action can adjust the environment workload or the user workload to reflect their interdependency. We will examplify its usage in modeling a malicious fault injection scenario in Section 8.2.3.

**Workload partitioning** (action 2 in Figure 5.1) partitions both the user workload and the environment workload to individual **per-actor workload**s ready for deployment. A workload from Chapter 4, either generated from a workload generator or a real trace, lists all the actions to perform during an experiment trial. We name such a workload **the unified workload**. The actions in a unified workload for a distributed system are performed from distributed points during the trial execution. It thus is mapped to the SUE or the execution environment through a number of distributed actors as discussed in Section 5.3.2 and Section 5.3.3. Accordingly, the workload lines associated with different actors need to be separated to prepare to for the distributed actors.

Along with the workload partitioning process, the unified workload may get tailored to a specific trial in two situations. In the first situation, the unified workload may contain some abstract information. It gets instantiated based on the properties configured for the specific trial. Because of the abstract, textual format of a unified workload, it can be reused for multiple trials of the same experiment or even across various experiments. When it is applied to a specific trial, some generally applicable but uninterpretable information contained in it needs to be translated to the concrete and meaningful parameters to be passed to the SUE or the environment to prepare for the actors. In the web-proxy example, each line in the unified user workload to access a web page has the format of `event(<timestamp>,<processid>,GET(<fileid>))`. The only parameter to the action `GET` should indicate the web address and client access port through which a file can be accessed. Since it is unknown in the unified workload which web servers are included in the SUE, the `fileid` in the unified workload is an uninterpretable identity to the web proxies. But when this workload gets partitioned for a specific trial, the `fileid` will be instantiated to a concrete web address and its client access port based on the trial properties. In specific, the configurations of the **host** model, the **component** model and the **SUE-to-testbed mapping** model provide the address and the client access port of each web server. In the trial configuration, we further configure that the web pages for which the identities are between `f1` and `f500` are located on server `S1` under the directory `test`. Thus, after being tailored to the trial configurations, the first workload line of the unified user workload, "event(1,U1,GET(f3))", is translated into the following lines in the per-actor workload file for the actor `A1` (assuming `A1` is associated to the workload process `U1`):

```
sleep 1
GET(http://skagen.cs.colorado.edu:8000/test/f3)
```

The second situation only happens for the environment workload. Since the environment workload is used to modify the trial execution environment, certain actions contained in it may cause unrecoverable effects so that the trial log files recorded on some testbed hosts get lost. To guarantee the integrity of the final experiment results, along with the workload partitioning process, whenever a loss-prone environment workload line is about to be translated into its corresponding per-actor workload, a data collection action is inserted into that per-actor workload first to precede the loss-prone action. The information on whether an action is loss-prone is configured by the software engineer for their specific trial execution environment. As the result of the extra data collection actions, during the trial execution phase, the intermediate log files are copied back to the master host before they may be lost.

The main action in the trial setup phase, **script generation** (action 3 in Figure 5.1), "compiles" the trial model configurations into a customized trial control system that will be used to automate the following three phases of the trial. We applied the generative techniques to this action, in which a generator is designed to implement the generation.

Generally speaking, "a generator is a program that takes a higher-level specification of a piece of software and produces its implementation [16]." It performs the following main tasks [16]:

- Checks the validity of the input specification and reports warnings and errors if necessary;
- Completes the specification using default settings if necessary;
- Performs optimizations;
- Generates the implementation.

In this context, the higher-level specification is the trial configurations. By applying the generative techniques introduced in Section 2.1.2, the work in manually programming control scripts for each trial is greatly saved by automatically generating them.

As summarized in Figure 5.1, the control system is made up of a master control script, a set of slave scripts and auxiliary files for each SUE component, the customized actor programs for each actor, and the communication library supporting inter-actor communications during trial execution. The master control script is generated to centrally coordinate the execution of the slave scripts and the actor programs to implement the system execution scenario described in the trial configuration. The slave scripts are called by the master script or the actor programs to control the execution and post-processing of the SUE components. The auxiliary files are normally just the customized system configuration files for the SUE components. All of them are generated by customizing the user-provided **ComponentType** configuration directives. The start/stop/post-processing slave scripts are generated from the **startScript**, the **stopScript**, and the **processing** attributes respectively. The configuration files are generated from the **config** attribute. Similarly, from the user-provided **ActorType** configuration directives, a customized workload interpretation program and a message receiving program are generated for each actor (a user actor or an environment actor) to actually perform its actions.

Besides these scripts and programs, the generator also generates the customized inter-actor communication library. The library provides the primitives for the actors to exchange messages and files during trial execution. The primitive implementation depends on the specific trial configuration since the real message and file transfer commands need the network address and possibly the directory of the target actor. The generated library thus keeps an **actor table** detailing the necessary information of each user actor and each environment actor, including its host address, host account, and its directory on the host. Whenever the inter-actor communication primitives are called, they will look up the actor table to get the necessary information for the target actors.

## 6.2 Trial Deployment

At this point, the engineer can conduct the trial by simply executing the trial master script. Figure 6.1 is the sequence diagram demonstrating how the different classes of distributed processes in the trial control system communicate during a trial deployment and execution scenario. The boxes across the top of the diagram represent the different process classes involved in a trial. Except the experimenter and the master script, all the other classes normally have more than one and often many instances distributed on the testbed because our framework is targeted at highly distributed systems. The dashed lines hanging from the boxes mean that they are available to be executed. The long, thin boxes on the dashed lines are activation boxes, during which the processes are active. The labels on the left side indicate the current activities, while the arrows between the dashed lines indicate the messages passing between the processes to execute requests.

The diagram shows the whole procedure automated by the trial control system, from the trial deployment until the experimenter receives and analyzes the trial results. It is obvious that the experimenter is only in charge of the configuration modeling and the final aggregated data analysis in this procedure without the need to meddle in the actions in between. The trial master script, instead, is in charge of communicating with the distributed processes to perform all the actions from trial deployment to data collection.

The first automated phase in the master script is the trial deployment phase. It covers three main activities that make the SUE available for experimentation (action 4 in Figure 5.1): the installation of the trial control system and the SUE, the trial bootstrap by activating the SUE components and the daemons in the trial control system, and finally the synchronization of all of the actors.

The installation of the trial control system and the SUE can be simply imple-

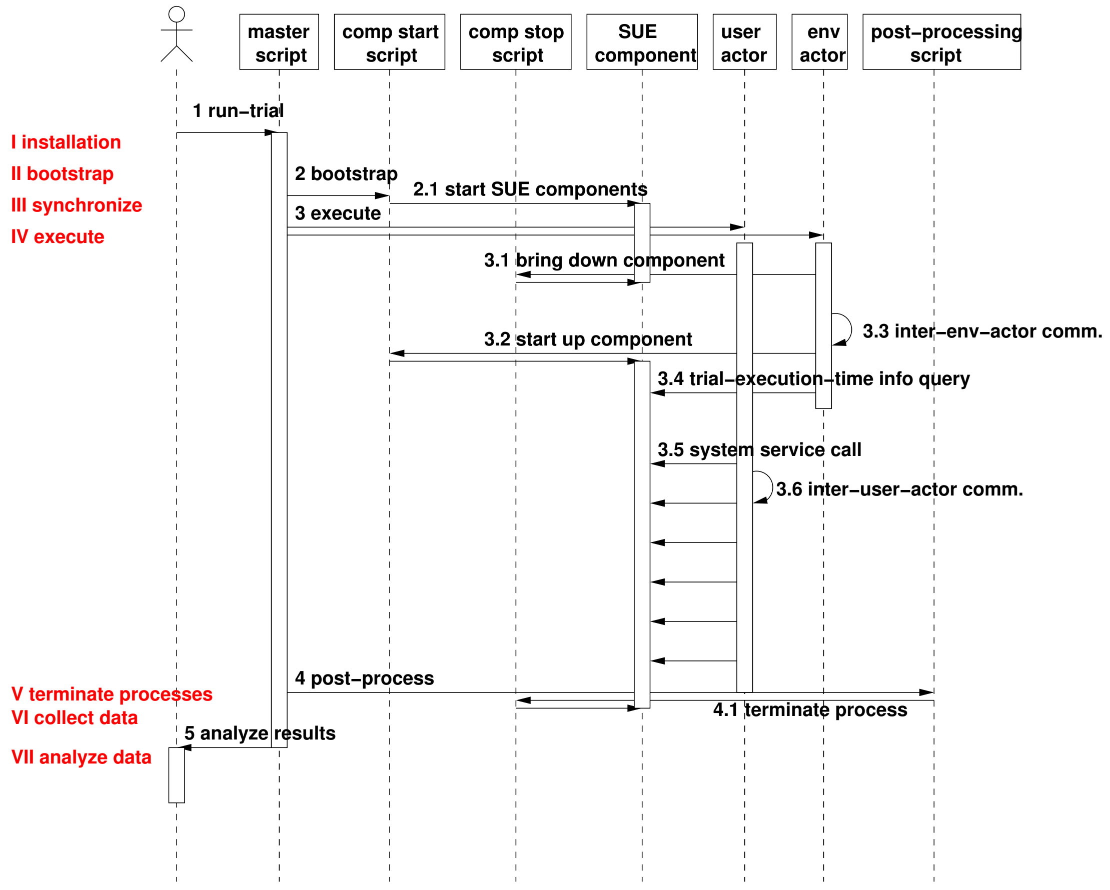


Figure 6.1: Trial Sequence Diagram

mented through file distribution. In specific, the trial control system is installed by transferring all the files except the master script to the remote hosts. Similarly, the SUE is installed by copying its system binary distributions. The master script has the knowledge of how the SUE components and the actors are mapped onto the distributed testbed. It thus deploys the per-actor workload and the actor programs for each actor, and deploys the slave scripts and the auxiliary files for each SUE component. Since the SUE binary distribution can be shared among the same type of SUE components, it only needs to be deployed once for each testbed host if that host hosts any component of this type. Likewise, the inter-actor communication library keeps the global information of the trial, it can be shared by all the actors. It only needs to be deployed once for each testbed host.

Besides the above generated scripts, files, and libraries, the trial control system also constitute some parts that is generally applicable to any experiment trial and is therefore directly provided by the trial automation framework. First, the data collection library is designed to solve the synchronization problem in recording trail log files. Because of the inconsistent clocks on the distributed testbed hosts, it is hard to aggregate and analyze the collected trial results containing the local times as the timestamp. The data collection library thus provides a primitive to direct a string to a specific log file with the trial virtual time stamped before the string. This trial virtual time is the time difference relative to the time when all the actors begin to process their workloads, i.e., the beginning of the trial execution phase. It thus is independent of the real world clocks on the testbed hosts. Second, to guarantee the reliability of the final trial results, the trial execution environment is expected to be constant. The trial control system thus possesses the trial monitoring mechanism to continuously check the status of the testbed. The monitoring mechanism can be implemented through any available distributed monitoring techniques. One copy of the monitor program is deployed onto each host together with other parts of the trial control system.

These deployed codes and data can be large and distributing them to a large number of hosts in a distributed testbed is costly. To make the trial automation framework efficient, all the codes and data can be packaged on a per-host basis and deployed in parallel. Other available software deployment techniques and tools discussed further in Section 2.1.4 can also be applied in this phase.

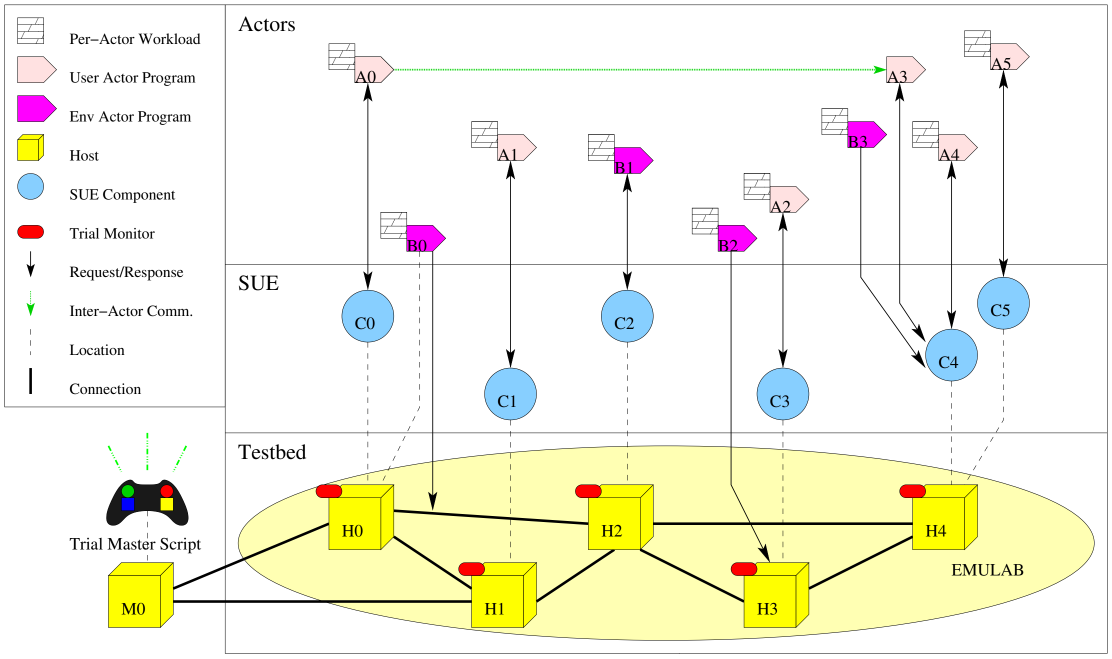


Figure 6.2: Post-Deployment Scenario of An Experiment Trial on Emulab

Figure 6.2 shows the scenario after an experiment trial of the Chord network example is deployed on an Emulab testbed. The master control script is on a local machine M0. M0 has the access to all the Emulab testbed hosts. After deployment, the start/stop/post-processing scripts of each SUE component are located on the component's host configured in the SUE-to-testbed mapping model. The multiple components of the same type located on a host share the same copy of the SUE binary distribution. The workload and the actor programs of each actor are located on the same host as the actor's associated entity. Since a user actor is always mapped to an SUE component, it is deployed onto the same host as the component, such as A0 through A5 in Figure 6.2.

The situation for an environment actor is a little more complicated since it can issue requests to change an SUE component, a testbed host, or even a testbed network link. However, either of these three possible entities can still be mapped to a testbed host. The host mappings for the first two entities are straightforward. Although a network link does not belong to any host, we only focus on how to modify the properities of a network link in this thesis. In other words, a network link can be mapped to any testbed host that has the ability to make changes to it, either through the testbed's own mechanism or through a certain traffic generator. For example, the environment actor B0 in Figure 6.2 will issue requests to the network link between host H0 and H2 during trial execution. It is deployed to H0 since a traffic generation program provided by the Emulab environment is installed on H0.

After the trial control system and the SUE are installed on the testbed, the SUE components and the control system daemons are activated to bootstrap the SUE. At this point, the SUE has been prepared to serve the trial execution. In the last synchronization activity, the master script communicates with all the distributed actors to learn the latency to communicate with them and to decide a future time to start the trial execution. By then, all the actors have been prepared to start the trial execution.

## 6.3 Trial Execution

Upon the completion of the experiment deployment phase, the trial has prepared for execution. It is time to start all the user actors to operate the SUE and all the environment actors to change the system execution environment based on the workloads. The master script efficiently communicates with all the actors scheduling them to begin processing their workloads at the trial start time (action 5 in Figure 5.1). One important issue to consider is how to start the many distributed actors simultaneously and keep them synchronized during the execution phase to follow the timestamps in the workloads.

The trial execution phase involves various of communications between the processes on the same testbed host as well as on different hosts. The master script first sends out an execution request to each user actor and each environment actor (interaction 3 in Figure 6.1). Upon receiving the requests, the actors wait until the calculated trial start time to start processing their per-actor workloads at the same time. The actors can have a variety of behaviors, some of which are shown in Figure 6.1. An environment actor may have the behavior of bringing down and starting up its associated SUE component. It implements such behaviors by translating the action like "BRINGDOWN" in the workload into a real shell command to execute the component stop script (interaction 3.1 in Figure 6.1) and the action like "STARTUP" workload line into a shell command to execute the component start script (interaction 3.2 in Figure 6.1). An environment actor can also communicate with other environment actors exchanging information (interaction 3.3 in Figure 6.1). It can also communicate with an SUE component directly through the system APIs to query information about the SUE (interaction 3.4 in Figure 6.1). For example, some malicious users of a distributed system may use system APIs to get information about system states. The attack plan can be scheduled accordingly to modify the system execution environment. We will examplify such interaction in the malicious fault injection scenario in Section 8.2.3.

In addition to those shown in Figure 6.1, an environment actor can also send out any other commands to change the execution environment, like a shell command to analyze the trial log files with the purpose of conditionally interrupting the trial execution based on the diagnostic data.

In contrast, the user actors only issue system service calls through the system APIs (interaction 3.5 in Figure 6.1). The system response to the service call may lead to inter-actor communications (interaction 3.6 in Figure 6.1). For example, if a user actor `A0` fails to get a response from the system, it may ask for help from another user actor `A3` by sending it a message.

## 6.4 Post-Processing

The trial execution phase may finish at the timeout configured in the trial configuration, or it may wait for all the actors to complete processing their workloads. The trial master script then causes the execution of the stop script for each of the SUE components. After all the components terminate, the post-processing scripts are executed against the component log files and the actor log files, the resulting diagnostic data are copied back to the master machine for further analysis. The testbed hosts are cleaned up if necessary (action 6 in Figure 5.1). We take his simplified approach in our trial automation framework, while more advanced distributed data analysis techniques introduced in Section 2.2.3 may be applied in the future to efficiently handle large quantities of experimental data in a distributed way.

## 6.5 Implementation

We have implemented the above automated trial process in our prototype Weevil. As described in Section 5.4, all the trial configuration inputs are programmed in GNU m4 by calling Weevil-defined declaration macros to instantiate the models. In other words, the declaration macros will define a set of property macros serving as properties of an experiment trial. These property macros are resolved during trial setup using m4's macro expansion. Weevil supports the engineer during trial setup by performing extensive checks on the syntax and consistency of the configurations, and by providing detailed error messages about any problems encountered.

The program used in the workload correlation activity to correlate the two workloads is provided by the software engineer. In our experiment, we borrowed the SSim discrete-event simulation library for its implementation. In specific, we created a simulation process to read each of the two workloads. The interdependency between the workloads was programmed through the event communication between the two reading processes.

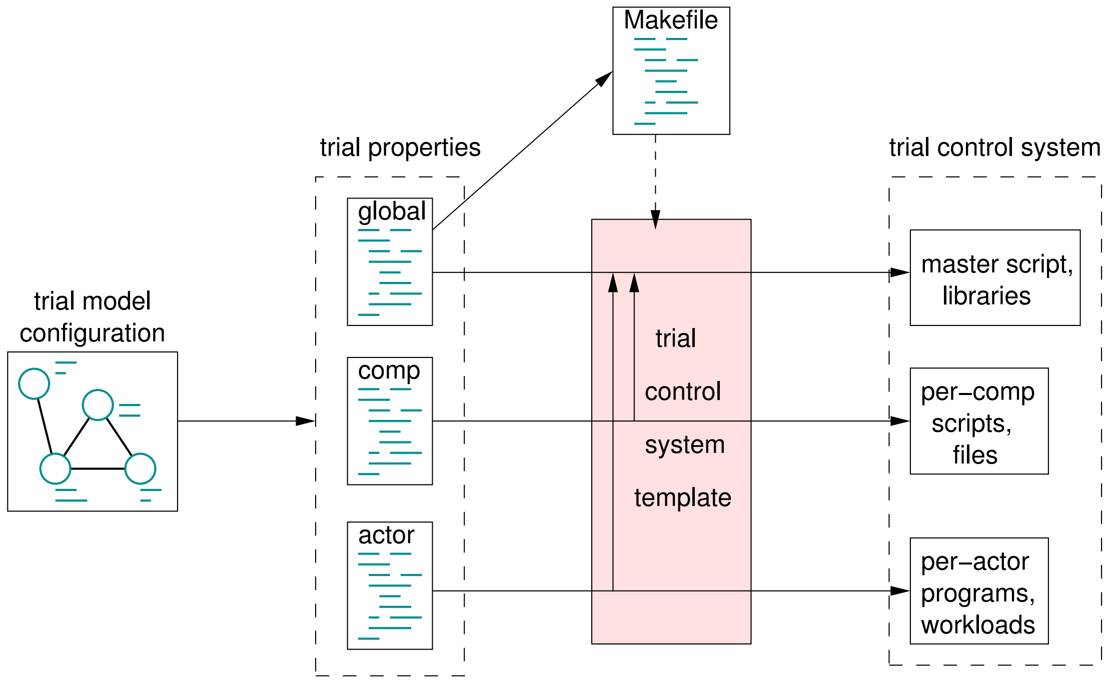


Figure 6.3: Weevil Trial Setup Process Using Macro Expansion

The translation from the configurations to the trial control system is a controlled process of m4's macro expansion. Figure 6.3 details how Weevil implements this process. The solid arrows stand for macro expansion, while the dashed arrow stands for script execution. Once they get approved, the trial configurations define three categories of property macros, the global ones, the per-component ones, and the per-actor ones. The global property macros describe the composition and properties of the trial, such as `theComponents`, which is the list of all the SUE components in the trial. The per-component property macros describe the properties of each component, such as its host address and its related component. Similarly, the per-actor property macros describe the properties of each actor, such as its associated entity and its host address.

Based on the trial global properties, a per-trial Make file having the knowledge of which files should be included in the trial control system is generated to control the rest of the trial setup process. The generated trial control system also includes three categories of files, the master script and the libraries, the per-component scripts and auxiliary files, and the per-actor programs and workloads. The trial global properties are unnecessary for the generation of the per-component and per-actor files. In our current implementation of Weevil, all the scripts in the generated trial control system are Bourne shell scripts.

The generated trial master script implements the efficient trial deployment, execution and post-processing using the parallel command execution tool **vxargs**[1] for its communication with the testbed hosts.

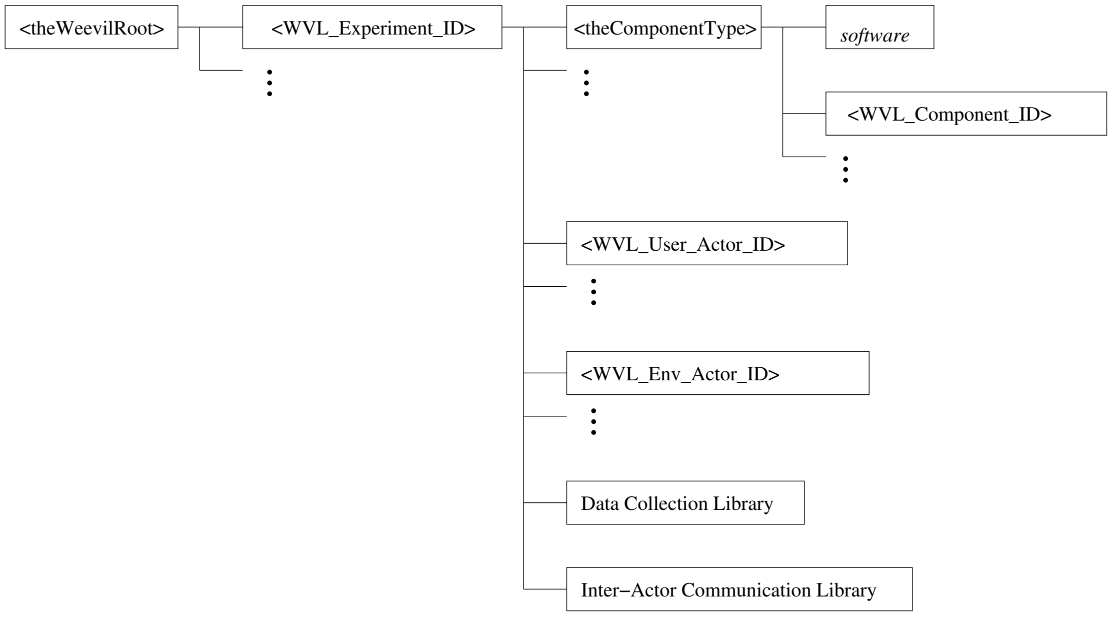


Figure 6.4: Weevil-Stuctured Directory Organization on Each Testbed Host

A trial is deployed onto the testbed and is well organized on each testbed host as shown in Figure 6.4. Every trial is allocated a separate directory under **theWeevil-Root**. The host-specific files including the data collection library and the inter-actor communication library are deployed into this trial directory. Under this trial's root directory, a separate workspace is allocated for each **ComponentType**, under which a single sub-directory `software` is allocated for the software binary distribution. Each

[1] http://dharma.cis.upenn.edu/planetlab/vxargs/

component has its own workspace under its corresponding **ComponentType** directory to store its scripts, configuration file and log files. Each **Actor** has its own workspace under the trial's root directory in which its actor programs, per-actor workload and log files are stored.

By estimating the round-trip time between the master host and each testbed host and based on the supported maximum number of parallel processes for a **vxargs** command, the master script intelligently calculates the time for it to finish communicating with all the actors. The trial start time is defined as the time right after all the actors are in communication and activated. In specific, Weevil calculates the trial start time using the following formula:

```
trial_start_time = current_time + (num_of_actors / parallel_num + 1) * max(round_trip_time)
```

When it is about to activate an actor, the trial master script calculates the distance from the current time to the trial start time on the master host, then passes this value as a parameter to the workload interpretation program asking it to sleep for the period indicated in the parameter. Beginning from the trial start time, the execution of the workload interpretation program for each actor is controlled through alternate sleep and wakeup operations based on its per-actor workload.

# Chapter 7

# Framework Reliability

In general, reliability is the “ability of a system to perform/maintain its functions in routine and also in different hostile or/and unexpected circumstances.”[1] Reliability support in the experimentation framework is very important. The experiments with distributed systems normally take place in complicated network environments and execute for a long time. The real world is not ideal. On Jun 27, 2006, we observed that 249 out of the 685 PlanetLab nodes were inaccessible by the experimenters. Notified and unnotified machine down time happens from time to time. Many error conditions arise during the experimentation process. If such accidents are not detected in time, they can distort experiment results and waste experiment resources. Thus, the reliability support for such unreliable, distributed testbed is essential. We dedicated significant effort to designing the reliability machanisms that make our trial automation framework resilient to failure in the testbed during experimentation. We are not targeted at providing smart incremental solutions. Instead, our goal is to optimize the linear parts in the experimentation process and avoid unnecessary repetitions.

Error conditions may be instant or may last for some time. A network may become very slow or out of service temporarily caused by unexpected network failure or traffic congestion at a peak period. The software engineer can wait for some time and try to continue the experiment in the same network later. Our trial automation

[1] http://en.wikipedia.org/wiki/Reliability

framework thus possesses the trial recovery mechanism to pick up the experiment trial from the latest successful state before the failure point. We will discuss this **recovery mechanism** in Section 7.1. However, not all error conditions are instant. Some remote machines may be out of service for several hours because of regular maintenance from the organization hosting the machines, while the software engineers are anxiously facing an approaching project deadline. They thus need to revise the trial configuration to bypass the failed machines and re-conduct the trial on a partially different testbed. We discuss the **partial-redeployment mechanism** that supports the process of bypassing the failure in Section 7.2.

## 7.1 Recovery Mechanism

A failed experiment trial is not totally useless. The purpose of our recovery mechanism is to make use of the successfully finished part of a failed trial to avoid later repetitions when recovering the trial. In particular, the generated trial master script that we introduced in Chapter 6 is logically broken into discrete work units that are isolated transactionally. The healthiness of the trial testbed is monitored along the whole trial process using the trial monitor daemons. Errors in the network are detected and reported as early as possible. Whenever an error is detected, the experiment trial is terminated, rolled back to the nearest point that is in a successful state, and started from that point so that the previous successfully finished work units of the trial need not to be repeated when an error occurs.

Errors can arise any time in a trial, and their effects on the experiment results are different. The trial execution phase is meant to study the performance of the SUE in operation. The system states along the whole execution phase can be reflected in the experiment results. Since the execution of a distributed system is a non-linear process, even if a failed operation is re-conducted successfully, the system state modification caused by the previous failed operation may still exist. Also, it is difficult to fully determine the effect of the error. The error condition may not be recoverable without initializing the system and re-executing the trial from the very beginning. Our recovery mechanism thus simply regards the whole trial execution phase as an individual work unit. Whenever an error is detected during trial execution, the master script stops the trial, cleans up the execution environment, rolls back to and continues the trial from the beginning of the trial execution phase. In contrast, the intermediate states during trial deployment or trial post-processing phase do not matter as long as all the tasks in the phases can be finished successfully at last. If an error arises during the trial deployment or post-processing phase, the experiment trial can be fully recovered as long as the failed operation can be successfully re-conducted. Our recovery mechanism thus separates the portions in the master script controlling these two phases into a series of work units, probably a work unit per remote communication. Whenever an error arises, the trial could be picked up and continued from whichever unit where the error arises.

## 7.2 Partial-Redeployment Mechanism

If an error lasts for some time, the trial automation framework regards the error as a lasting error. The recovery mechanism no longer works since it cannot avoid the error. To continue with the experiment, a revised trial has to be conducted instead on a testbed with the failed hosts bypassed. Figure 7.1 shows the overall process for the revised trial. The legend of the figure is almost the same as that of Figure 5.1 except that the dark ovals in Figure 7.1 represent the existing files left by the failed trial instead of the user inputs in Figure 5.1.

A revised trial is also automated through a five-phase process similar to a regular trial. However, instead of starting from scratch, the failed trial provides some files that may be helpful in the revised trial (the dark ovals in Figure 7.1) including its trial configuration file, its unified and per-actor workloads, and its generated trial control system. Moreover, if the failed trial has reached the trial deployment phase, some

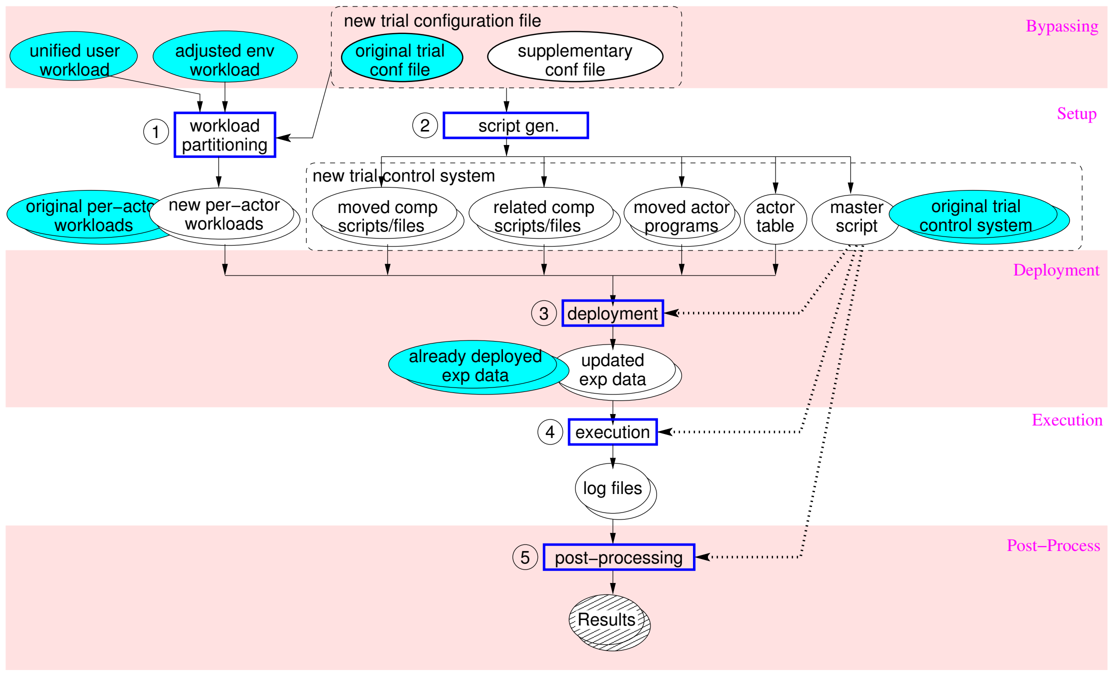


Figure 7.1: Automated Partial Re-generation and Re-deployment Process to Bypass Failed Hosts

experiment data also exist on the testbed.

The purpose of our partial-redeployment mechanism is to replace the failed hosts with backup hosts, at the same time avoiding unnecessary repetition in the replacement process. Although the testbed configuration in the new trial is different from that in the failed one, thanks to the modularization of our trial configuration models, the configuration difference between the failed trial and the revised one can be as simple as several lines of new host configurations. Thus, instead of a new configuration file containing the full configuration modeling, only a supplementary configuration file including all the differences is necessary for the revised trial. We thus call the first phase **bypassing** phase instead of **modeling**.

The **trial setup** phase based on the supplementary configuration file does not include as much work as that of a brand new trial discussed in Section 6.1. Since the per-actor workloads and the trial control system generated for the failed trial still exist, only those scripts and libraries affected by the configuration differences need to be re-generated. The re-generated parts then replace their out-of-date copies to serve the trial as indicated by the overlap of ovals in Figure 7.1. Our model-based generative approach provides a natural way to isolate the affected files from the unaffected ones of the trial control system. In specific, the affected parts that need to be updated include:

- Those of the SUE components that were mapped onto the failed hosts: Because of the replacement of their hosting machine, those components that were located on the failed hosts (illustrated by the SUE-to-testbed mapping configuration) have to be transferred to the backup hosts. Accordingly, their start/stop/post-processing scripts and auxiliary files need to be re-generated to reflect the host changes.

- Those of the actors that were mapped onto the failed hosts or to the trial entities related to the failed hosts: In turn, the actors (either user actors or environment actors) associated to the failed part of the testbed or to the entities located on that part (illustrated by the actor mapping configuration) have to be transferred to the backup hosts. Not all the files of those actors need to be re-generated since some workloads and actor programs are independent of testbed configuration. Only those workloads and actor programs using testbed properties should be re-generated with the backup hosts replacing the failed hosts. Note that the workload correlating action (action 1 in Figure 5.1) does not exist in the revised trial. This is because the workload correlating action only uses the actor mapping configurations, thus the testbed configuration changes will not affect the generated adjusted environment workload.

- Those of the correlated components: The transferred components may have been referenced by other SUE components configured in the **ComponentRelation** attribute of the SUE model. Because of the host changes to the transferred components, some properties of these components change. Accordingly, the start scripts and configuration files of the correlated components, in which the transferred components get referenced, need to be re-generated.

- The actor table of the inter-actor communication library: Similarly, the transferred actors may be contacted by other actors during trial execution. The inter-actor communication library keeps the host information of all the actors in the actor table. This table thus needs to be updated with the properties of the new hosts replacing those of the replaced hosts.

- Trial master script: At last, the master script needs to be re-generated. This complementary master script is different from the ordinary trial master script in that it only deploys the above re-generated files and those experiment files that have not been deployed in the failed trial. Whereas, it controls the trial execution and post-processing phases in the same way as the regular master script.

Upon the completion of the trial setup phase, the remaining four phases are exactly the same as those of a regular trial since all the revisions have been made in the trial control system. By executing the trial master script, the revised trial is executed and post-processed on the new testbed. The partial-redeployment mechanism is cumulative. If a revised trial fails again, the same method can be applied with a new supplementary configuration file used.

## 7.3 Implementation

In our prototype Weevil, we have implemented both the recovery mechanism and the partial-redeployment mechanism to support our large-scale experiments. As we have mentioned in Section 6.5, we used the parallel execution tool **vxargs**[2] to implement the communication between the trial master script and the distributed SUE component scripts and actor programs. **vxargs** executes based on a task file listing the commands for all the parallel processes it will manage. When a **vxargs** statement is executed, it keeps the record of the status of each parallel process it manages. Whenever a process is found to be abnormal, the **vxargs** terminates with the identities of the abnormal process listed in a file named **abnormal_list** and the correctly finished processes listed in a file named **finish_list**.

We implemented our recovery mechanism using this feature of **vxargs**. As shown in Figure 7.2, we divided the Weevil trial master script into a sequence of work units: trial directory setup, system installation, SUE component startup, trial monitor daemon startup, actor synchronization, trial execution, trial process termination, log file analysis, data collection, and testbed cleanup. Each work unit is implemented through a **vxargs** statement to operate on the distributed testbed hosts in parallel. Weevil keeps a flag for each work unit indicating whether that unit has been successfully finished. Thus

[2] http://dharma.cis.upenn.edu/planetlab/vxargs/

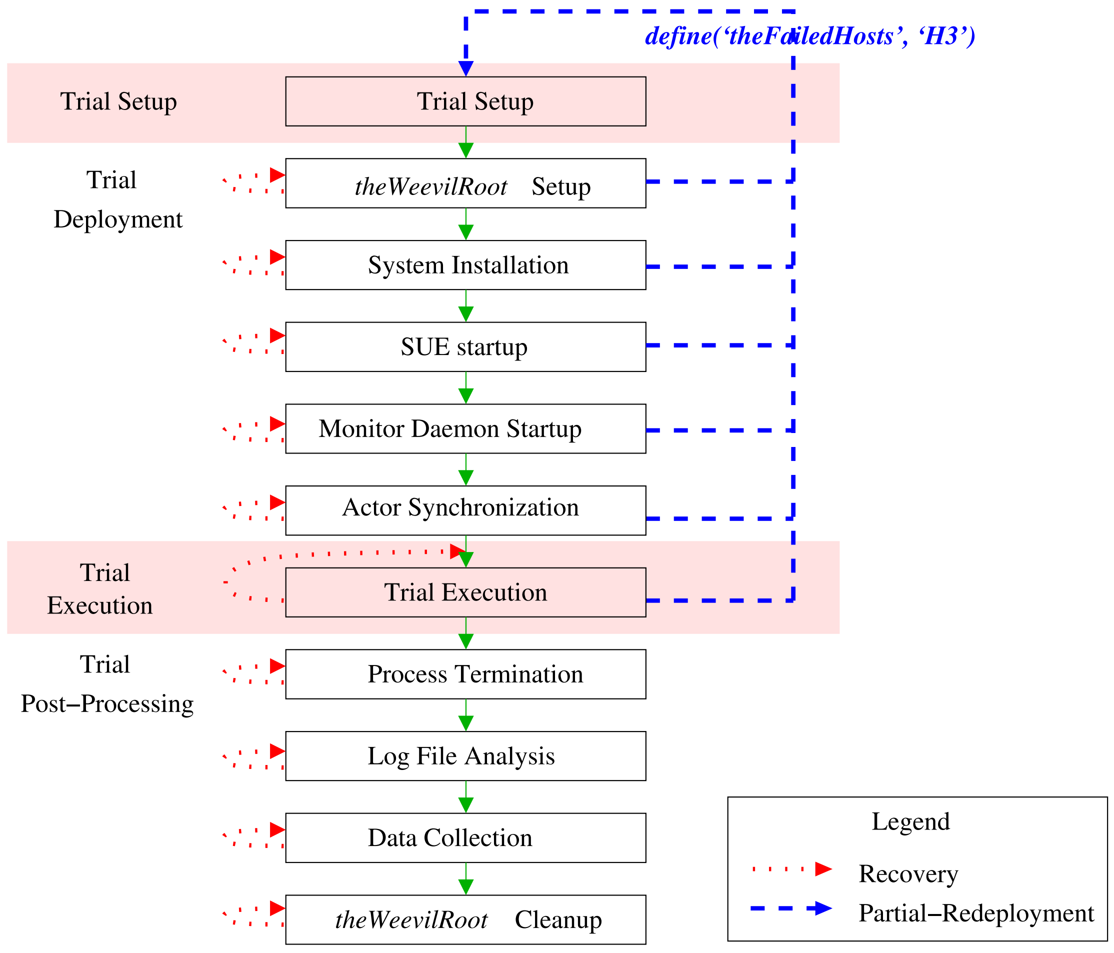


Figure 7.2: Weevil Reliability Mechanisms

every time Weevil re-conducts a trial, all the work units with the flag set to 1 (meaning finished) will be skipped to avoid repetition. Shown as the dotted arrows on the left side in Figure 7.2, if the execution of a **vxargs** statement fails, based on the status of each paralell process, Weevil is able to update the task file of **vxargs** and set the flag of the work unit to -1 (meaning partly finished). Then when the trial is re-conducted, **vxargs** only executes the unfinished tasks in the last trial run. The trial execution work unit is different from the others: Instead of updating the task file, Weevil keeps its task file unchanged to re-run the whole execution phase.

If an error happens multiple times, Weevil regards it as a lasting error. Shown as the dashed arrows on the right side in Figure 7.2, when the lasting error happens, a single-line m4 macro indicating the failed hosts is returned back to Weevil. Based on the trial model configurations, Weevil maps each failed host to an unoccupied testbed host (the backup host) and generates a supplementary configuration file. Suppose that H50 is a configured but unoccupied testbed host, and the SUE component C3 and C53 are both mapped onto H3 originally, the generated supplementary configuration file will have the following contents:

```
define(‘theDownHosts’, ‘H3’)
define(‘theBackupHosts’, ‘H50’)
define(‘WVL_Host_’H3‘_replace’, H50)
WVL_SYS_ComponentHost(‘CHMap3’, C3, H50)
WVL_SYS_ComponentHost(‘CHMap53’, C53, H50)
```

Similar to Figure 6.3, Figure 7.3 shows how Weevil re-generates the updated parts of the trial control system based on the trial configuration file in the failed trial and the supplementary configuration file. The two configuration files make up the configurations for the new trial. The property macros defined by them can be divided into four categories. Three of them are the same as those for an ordinary trial, the other one is the set of macros listing the trial entities affected by the failed hosts, like `theTransferredComps`, `theTransferredActors`, and `theCorrelatedComps`.

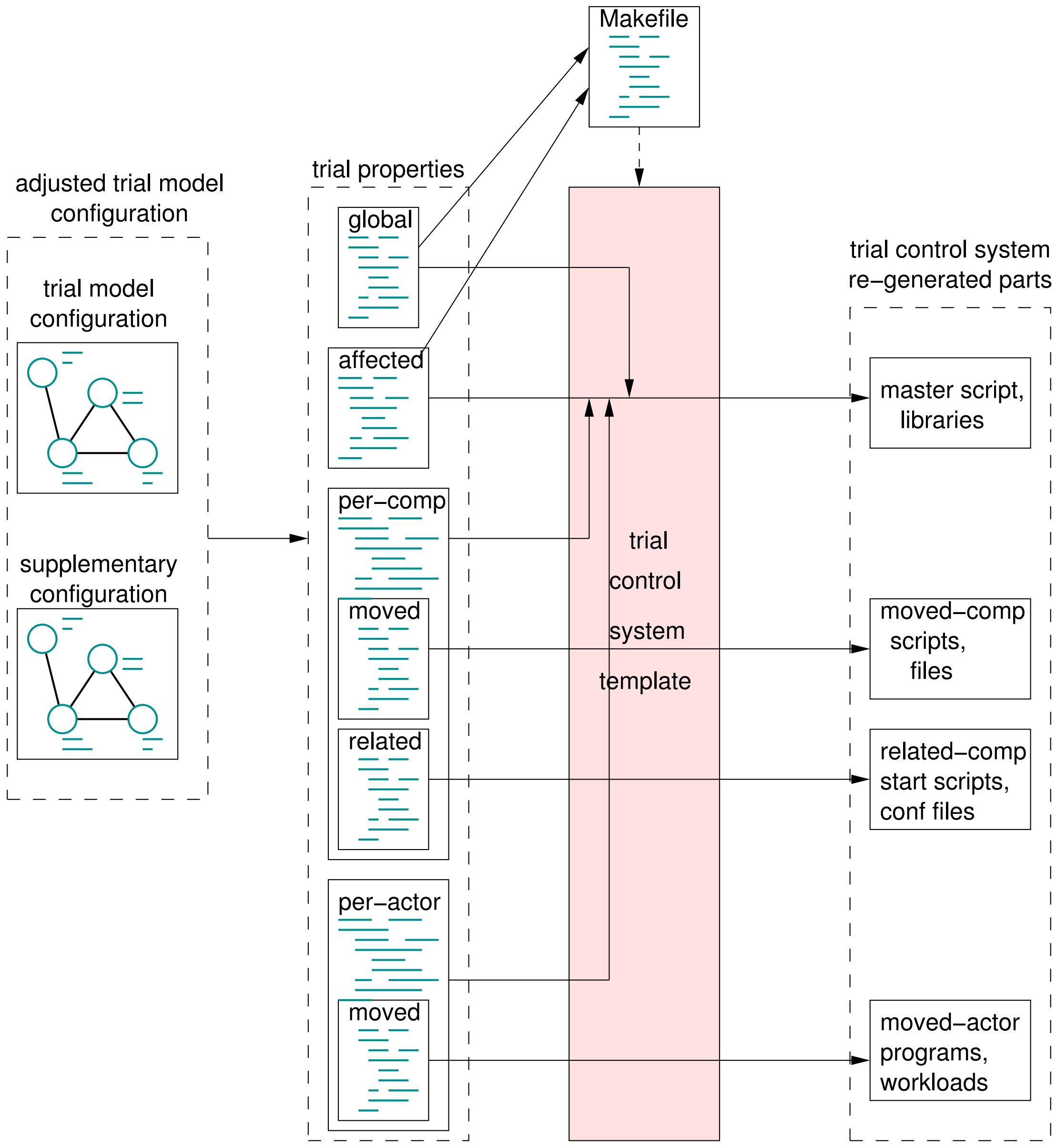

Makefile
trial properties
adjusted trial model configuration
trial model configuration
supplementary configuration
global
affected
per–comp
moved
related
per–actor
moved
trial control system template
trial control system re–generated parts
master script, libraries
moved–comp scripts, files
related–comp start scripts, conf files
moved–actor programs, workloads

Figure 7.3: Trial Setup in Weevil to Bypass Failed Hosts

These macros together with the trial global properties can be applied to generate a per-trial Make file containing the knowledge of which files should be updated in the trial control system is generated to control the rest of the revised trial setup process. The updated files are divided into four categories, the master script and the libraries, the per-moved-component scripts and auxiliary files, the per-moved-actor programs and workloads, and the per-correlated-component start scripts and auxiliary files.

# Chapter 8

# Experience and Evaluation

To evaluate the ability of our trial automation framework in conducting experiments with various distributed systems, especially highly distributed systems, on different large-scale testbeds for wide-ranging experiment scenarios, we have conducted thorough evaluation using our implemented prototype, Weevil. Our evaluation activities, including both case studies and meaningful experiments, can be divided into five categories based on the five research questions we were trying to answer:

- **Fidelity of the generated workloads:** Is the simulation-based workload generation approach faithful to specific problems and scenarios?
- **Versatility of the configuration models:** Are our trial configuration models applicable to a wide variety of distributed systems and testbeds? And can our actor behavior modeling approach capture different experiment scenarios?
- **Scalability of the deployment and execution mechanisms:** Is our framework capable of handling large-scale experiments on wide-area network testbeds?
- **Utility of the automation features:** Does our model-based generative approach provide effective cost savings in automating the experimentation process?
- **Support of the unreliable testbeds:** Are the reliability mechanisms that

our framework provides able to correctly and efficiently support experiments on unreliable testbeds like PlanetLab?

In each of the following sections, we describe our experience with respect to each individual question.

## 8.1 Fidelity of the Generated Workloads

To verify that the simulation-based workload generation approach can faithfully create realistic workloads, we developed workloads representing the web-cache client behavior described in a study on cooperative web caching [62]. According to the study, cooperative caching improves cache hit rate rapidly with smaller populations, while it is unlikely to provide significant benefit for larger populations. The study was based on traces collected from multiple live proxies at the University of Washington and Microsoft Corporation, data that are usually difficult to obtain.

Based on the analysis provided by the trace study, we set up the workload generator by modeling and simulating the behavior of several actor groups, each one representing an independent organization in the study, such as a different department at school. The actors within one group use the same proxy and tend to have similar interests, resulting in a higher level of intra-group communication than inter-group communication. Thus, the population of actors in a scenario with fewer groups is more homogeneous overall in its web-access behavior. To model the intra-group and inter-group communication, the behaviors modeled in the simulation program include level 2 actor behaviors, which are dependent on each other. We generated several different workloads by executing the simulation program with different actor population configurations for a three-group organization and a six-group organization respectively.

The workloads were then applied to a distributed web-caching system made up of several Apache servers and three or six Squid proxies, depending on the number of actor groups modeled in the workload. We will discuss some modeling issues of this system later in Section 8.2.2. In this specific experiment, we used a local-area testbed consisting of six hosts running Linux or FreeBSD operating system. Using a local-area testbed is reasonable in this case, since the cache hit rate is unrelated to network latency.

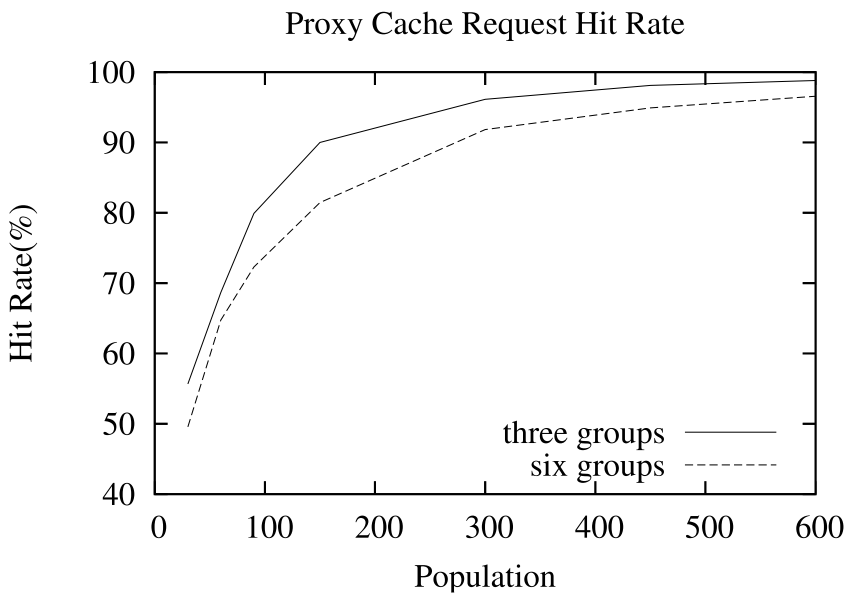


Figure 8.1: Cache Hit Rate vs. Client Population

Figure 8.1 shows the hit rates for experiment trials with three actor groups (proxies) and experiment trials with six actor groups (proxies). Each of the plotted lines has an inflection point with a steep increase in request hit rate below and a shallower increase above. The experiments with three groups have higher hit rates, which results from their more homogeneous behavior.

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

## 8.2 Versatility of the Configuration Models

Our first concern in evaluating the ability of the model-based generative approach in automating experimentation with distributed systems is whether its configuration models are easily applicable to a wide variety of testbeds, distributed systems, and experiment scenarios. We therefore first carried out some case studies, in which we used the models to describe some representative testbeds, systems and their combinations. Through our configuration modeling experience, we found that the trial configuration models can capture all the necessary characteristics of the experiment trials we studied in a modularized way. Changes in one module have very few or no effects on the others. And the models are able to accommodate the specific features of different testbeds and distributed systems. Our discussions of these case studies are independent of specific experiments. Instead, we discuss how Weevil handled the specific features brought by representative testbeds and systems.

We also carried out some meaningful experiments with a distributed service CFS to study its performance and robustness. In specific, we modeled four different fault injection scenarios by modeling different actor behaviors and studied the CFS under these scenarios. Through our actor behavior modeling experience, we found that the combination of the workloads and the actor programs can flexibly capture wide-ranging actor behaviors. This ability in turn can meet the need of studying the SUE in different scenarios.

### 8.2.1 Testbed Modeling

As discussed in Section 2.1.3, a testbed for distributed system experiments can be a local-area network, a wide-area overlay network like PlanetLab, or even an emulated network such as Emulab. No matter which kind of testbed to be used in an experiment trial, our testbed model regards and models it as a collection of distributed hosts that are accessible through user-level remote shell access. The underlying infrastructure of the testbed is transparent to the experimenter. Moreover, our trial automation framework only configures a testbed ready for use without actually setting it up. The testbed setup process, including machine installation, connection, and operating system environment setup, is normally well supported by testbed providers.

Using the testbed conceptual model, we configured a local-area network testbed, a wide-area overlay network testbed on the PlanetLab, and an emulated network testbed on the Emulab for the experiment trials with the same distributed system in need of three machines to deploy. The portion of the testbed configuration for each of the three testbeds is shown in Figure 8.2.

As we expected, our experience shows that the testbed modeling of each of the three types of testbeds reflects no big differences from one another. The only difference arose in the configuration of the Emulab testbed. In this configuration, other than its required attributes, each host is also described with two extra attributes, the `alias` and the `internal_ip` address of each host. This is because of the particularity brought by the **Emulab control network**.

As shown in Figure 8.3[1] , there exist two types of networks in Emulab, the Emulab control network and the user-configured experiment network, i.e., the testbed. The view from inside of the testbed is different from that of a host out on the Internet or from the Emulab master host “users.emulab.net”. From an outer host, a testbed host needs to be accessed via its canonical Emulab name (e.g., “pd12.emulab.net”) or its DNS name that is assigned when the experiment testbed is created (e.g., “node2.foo.testbed.emulab.net”). Then the testbed hosts are accessible through the fixed 100Mb control network link (the solid links in Figure 8.3). However, from a testbed host, if either of the above two names of another testbed host is used, the connection between the two hosts will still go through the Emulab control network instead of the experiment network that the experimenter

[1] http://www.emulab.net/tutorial

```
define(‘hrange’, ‘WVL_SYS_Range(‘H’, 0, 2)’)
WVL_SYS_Testbed(‘Bed_0’, ‘hrange’)
WVL_SYS_HostType(‘Linux’)
WVL_SYS_Host(‘H0’, ‘skagen.cs.colorado.edu’, ‘ywang’, ‘/tmp/weevil’, ‘/bin/sh’,
  ‘/usr/java/j2sdk1.4.2_07/bin/java’, ‘Linux’)
WVL_SYS_Host(‘H1’, ‘leone.cs.colorado.edu’, ‘ywang’, ‘/tmp/weevil’, ‘/bin/sh’,
  ‘/usr/java/j2sdk1.4.2_07/bin/java’, ‘Linux’)
WVL_SYS_Host(‘H2’, ‘serl-back.cs.colorado.edu’, ‘ywang’, ‘/tmp/weevil’, ‘/bin/sh’,
  ‘/usr/java/j2sdk1.4.2_07/bin/java’, ‘Linux’)
```

(a) Local-Area Network

```
define(‘hrange’, ‘WVL_SYS_Range(‘H’, 0, 2)’)
WVL_SYS_Testbed(‘Bed_1’, ‘hrange’)
WVL_SYS_HostType(‘PlanetLabHost’)
WVL_SYS_Host(‘H0’, ‘planetlab1.cs.colorado.edu’, ‘colorado_weevil’, ‘/home/colorado_weevil/weevil’,
  ‘/bin/sh’, ‘usr/java/j2sdk1.4.2_07/bin/java’, ‘PlanetLabHost’)
WVL_SYS_Host(‘H1’, ‘planetlab3.csail.mit.edu’, ‘colorado_weevil’, ‘/home/colorado_weevil/weevil’,
  ‘/bin/sh’, ‘/usr/java/j2sdk1.4.2_07/bin/java’, ‘PlanetLabHost’)
WVL_SYS_Host(‘H2’, ‘planetlab6.millennium.berkeley.edu’, ‘colorado_weevil’,
  ‘/home/colorado_weevil/weevil’, ‘/bin/sh’, ‘/usr/java/j2sdk1.4.2_07/bin/java’, ‘PlanetLabHost’)
```

(b) PlanetLab

```
define(‘hrange’, ‘WVL_SYS_Range(‘H’, 0, 2)’)
WVL_SYS_Testbed(‘Bed_2’, ‘hrange’)
WVL_SYS_HostType(‘EmulabHost’)
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_Host(‘H’i, ‘node’i‘.example.Weevil.emulab.net’, ‘ywang’,
  ‘/tmp/weevil’, ‘/bin/sh’, ‘/usr/java/j2sdk1.4.2_07/bin/java’, ‘Linux’)’, hrange)
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_HostProp( ‘H’i, ‘alias’, ‘node’i)’, hrange)
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_HostProp( ‘H’i, ‘internal_ip’, ‘10.1.1.’eval(i+2))’, hrange)
```

(c) Emulab

Figure 8.2: Testbed Modeling Case Studies

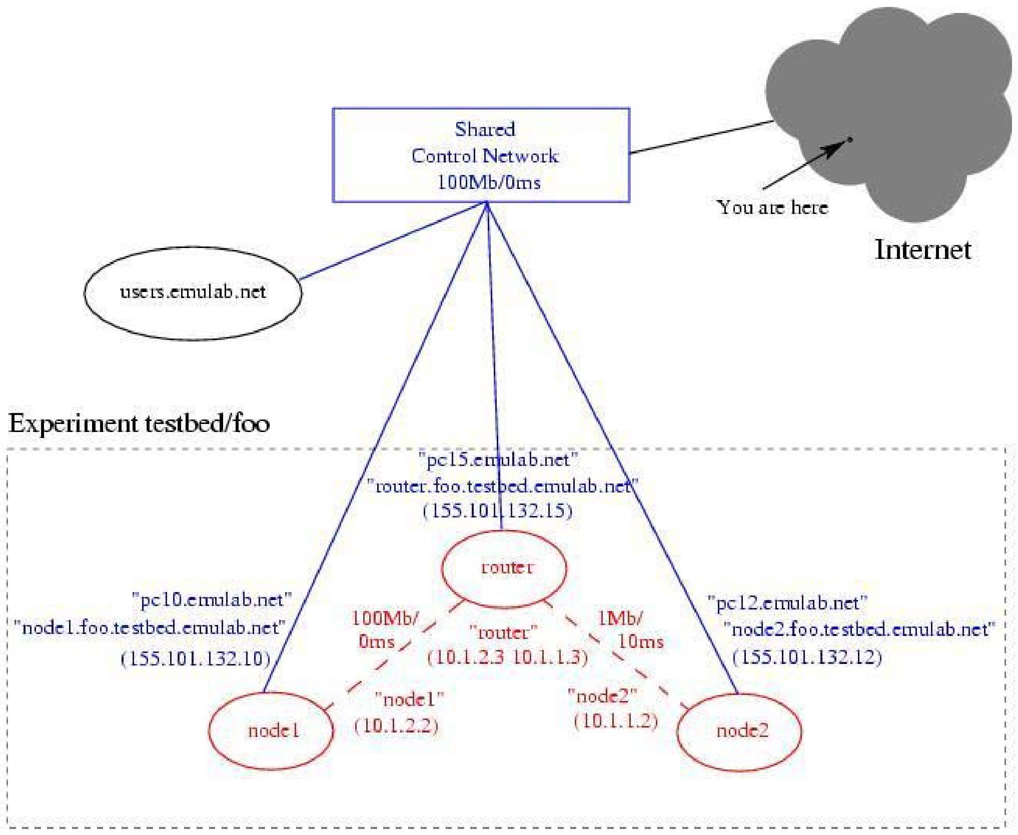


Figure 8.3: Emulab Control Network Interface (Blue Solid Lines) to an Experiment Network (Red Dashed Lines). See http://www.emulab.net/tutorial.

expects. Emulab currently handles this problem by requiring each SUE component to be connected by another component through the unqualified alias (e.g., "node2") or the internal ip address (e.g., "10.1.1.2") of its testbed host. Then all the traffic between the components will go through the experiment testbed (the dashed link in Figure 8.3). Most distributed systems include options enabling the user to explicitly specify which network interfaces to use for peer communication. Thus, the two additional Emulab host attributes `alias` and `internal_ip` can be used in the **ComponentType** configuration to specify the correct network interface for each SUE component.

Weevil is able to conduct experiment trials on any one of the three testbeds by including the corresponding testbed configuration. Changes in the testbed configuration have little or no effect on the other parts of configuration modeling.

### 8.2.2 SUE Modeling

One of our purposes of applying the model-based generative approach is to make the trial automation framework generally applicable to experiments with a variety of distributed systems. To evaluate the framework's breadth of applicability, we modeled and ran experiments on six, quite different highly distributed systems: Siena [10], a publish/subscribe service implemented through a network of servers; Elvin [53], a publish/subscribe middleware system that supports the federation of servers; MobiKit [8], a mobility framework for distributed publish/subscribe systems that is implemented through a proxy-client mechanism; Freenet [15], a peer-to-peer file sharing system; Chord [54], a distributed value/location lookup service based on distributed hashing; and a composite web-cache system made up of Squid proxies [27] and Apache web servers [40].

The six systems provide a representative sampling from the broad spectrum of highly distributed systems. Each has unique characteristics that requires special consideration when configuring experiment trials. Below we illustrate the ways in which

Weevil was able to accommodate some of these system-specific features for each of them.

**Siena** implements a publish/subscribe service through a hierarchy of servers. The hierarchy of Siena servers is established by starting up each server with a command-line parameter that indicates the address and packet receiving port of its parent server. And its parent server needs to be ready before each server can start up itself. Each Siena client accesses the service through a local server. It injects subscription requests and publications into the SUE using Siena's Java API.

Siena servers' hierarchy structure complicates the experiment setup because it requires each server start script to adapt to the testbed and the topology. Without our trial automation framework, we have to write different start script for different component in an experiment. A specific component also needs to modify its start script to reflect a different testbed and topology configuration. We also need to control the start-up order of these start scripts.

Through its SUE conceptual model and the SUE-to-testbed mapping, Weevil simplifies this situation by allowing the common start script of a type of components to be parameterized by the properties of the components as well as the testbed. We therefore handled this configuration issue as follows: (1) we declared a single **ComponentType** "SienaServer", defined all components as an instance of this type, and specified the start-up **Order** based on the server topology; (2) we assigned each component a "port" property through the **ComponentProp** declaration function; (3) we configured the Siena hierarchy by defining a "parent" **ComponentRelation** between components; (4) we mapped each component to a testbed host; and (5) we used the property macros defined in the above configurations in the start script defined for the **SienaServer** component type, so that for each Siena server, the "parent" parameter of the start script is automatically bound to its parent's testbed machine address and packet receiving port. Using this straightforward approach, trial setup can be easily adapted for different components, testbeds and hierarchical topologies by only modifying the configuration of

**Component**, **Testbed** or **ComponentRelation**, respectively.

**Elvin** is another publish/subscribe middleware system that supports federation of servers. We only had access to the its Linux binary distribution, but one of our testbeds was made up of machines running FreeBSD. If an experiment trial of Elvin is conducted on this testbed, we can 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 excluding the **ComponentType/HostType** mapping in the trial configuration file. We also needed to configure the start script of the component type **ElvinServer** to access Elvin's pre-installed binaries, which was easily accomplished by setting a system-specific property called `executionPath` for each testbed host, and using this property macro in the start script.

**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.

We declared two **ComponentType** to stand for the mobility proxies and the publish/subscribe servers respectively. Each of them was modeled just like a standalone system of itself. The only add-on is a list of component **Relation**s, each of which defines which proxy is the nearby proxy of each server.

The most significant problem posed by MobiKit is that the mobile client is not fixed to a Mobikit proxy or a publish/subscribe server during experiment execution while Weevil requires each actor bound to a component through a fixed actor mapping.

We were able to solve this additional binding problem by having several actors representing a single mobile client. In specific, a mobile client at a different location was represented by a different actor which is mapped to the nearby Mobikit proxy in that location. For example, if the mobile client **C0** moves from location A to B, it is represented by two corresponding workload processes, **P00** and **P01**. The workload entries of **P01** have larger time stamps than those of **P00**, representing the requests sent by the same client after it moves from A to B. Thus, the workload of the client is actually the integration of all the workload entries from its representative simulation processes. The movement of the client is represented by including `movein(<source>)` and `moveout` entries in the workload. For example, the last workload entry of **P00** is "moveout", and the first workload entry of **P01** is "movein(P00)", representing the client moves out of A and moves to B. The `<source>` macro is expanded based on the user actor mapping and component configurations during Weevil's workload partitioning phase providing the information about the previous proxy that the client is bound to. The actor programs were implemented to translate "movein", "moveout", and publish/subscribe requests to the corresponding system service calls. During trial execution, based on the workload, the actor representing **P00** terminates its execution before the actor representing **P01** begins, just like what a mobile client would do.

**Freenet** is a peer-to-peer file-sharing system. A feature of Freenet, which is quite common among other distributed systems, is that its components are controlled by configuration files rather than command-line parameters. This requires Weevil to customize a configuration file on a per-component basis.

As shown in Figure 5.2, **ComponentType** contains a **config** attribute that is the content of its configuration file. This content is retrieved during script generation using m4's include mechanism. Thus, we added macros into the default Freenet configuration file and filled in the file name as the **config** attribute of the component type **FreenetNode**. The following snippet from our template Freenet configuration file shows how the `listenPort` parameter is set by Weevil.

```
listenPort = ‘WVL_Component_’WVL_Component_ID‘_ListenPort’
```

Each **FreenetNode** component has a `ListenPort` property assigned to it that is stored in a property macro of the form: `WVL_Component_<ID>_ListenPort`. During setup, the macro `WVL_Component_ID` is defined in turn to the identifier of each component currently being processed. So, the macros on the right-hand side of the assignment expand to be the listen port for the component being processed. Other component-specific parameters in the template configuration file were set similarly. As the result, the template configuration file was customized for each component. And Weevil gives each of them a name `<WVL_Trial_Name>_<WVL_Component_ID>.conf`. Similarly, the component's start/stop commands were customized by including macros in the **startScript/stopScript** attribute of **FreenetNode**. For example, an argument of a Freenet node's start command is the location of its configuration file. Since the configuration file has been generated and deployed to the component's **theCompPath** directory on the corresponding testbed host before trial execution, we specified the argument by including the following argument in the start script:

```
theCompPath/WVL_Trial_Name‘_’WVL_Component_ID.conf
```

**Chord** implements a distributed lookup protocol that allows its clients to lookup a location (node) associated with a key. A number of Chord instances can be correlated to make up a peer-to-peer Chord network. For the Chord binary distribution we used, the Chord instances are configured by their command-line parameters and can be correlated by indicating the host name and port of a bootstrap node in their start commands. The bootstrap Chord instance needs to be started up first.

We handled the bootstrap issue the same way as Siena assuming that all the components have the bootstrap node as their parent. In addition to the bootstrap node parameter, a number of other Chord command-line parameters are about the resources used for the Chord node execution, such as the path to the database used by the node. It should be a valid path on each testbed host. Another parameter is the network interface used by the node. As we have discussed in Section 8.2.1, to use the Emulab testbeds, this parameter needs to be explicitly spcified as the internal ip address of the testbed host on which the Chord node is located.

**Squid/Apache** Squid is a full-featured caching web proxy, which in this composite web-caching system serves Apache servers. Two component types, **ApacheServer** and **SquidProxy**, were defined in the SUE model. Binaries for Squid and Apache are available for a number of different platforms. Through this example, we highlights Weevil's ability in handling heterogeneous testbeds.

Assuming a testbed used for an experiment of this system is made up of Linux machines as well as FreeBSD machines. Weevil supports the deployment of a single type of components onto a heterogeneous testbed through the **type** attribute of each host. When the master script is deploying component software to the hosts, the **ComponentType/HostType** mapping is used to select the appropriate source directory of the binary distribution on the master host for each component type based on the type of each testbed host to be copied to.

The Squid/Apache system 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 program, which parses the workload lines and hands off the execution of the actions to the specified program.

### 8.2.3 Experiment Scenario Modeling

Even if the testbed and the SUE are both determined, we can still model different actor behaviors to study the SUE under different usage scenarios and environment variations. To evaluate our trial automation framework's flexibility in modeling different experiment scenarios, besides the many experiments discussed in the later sections to study the SUE under constant environments, we also modeled four different fault injection scenarios and ran robustness testing experiments with a wide-area cooperative storage service, CFS [17], on the Emulab testbeds. In specific, we conducted four experiments, each of which consists a set of trials and uses a different fault injection scenario model. The four scenarios we modeled are the simultaneous fault injection scenario, the continuous fault injection scenario, the reactive fault injection scenario, and the malicious fault injection scenario.

**Simultaneous Fault Injection**

The core of the CFS software consists of two layers, DHash and Chord. The Chord layer, as the backbone of the CFS service network, is to locate the Chord node responsible for a storage block. A Chord network achieves the ability to regain consistency after a large percentage of Chord nodes fail simulataneously [54]. But since a Chord network only keeps one copy for each data block, half of the blocks would lose if half of the Chord nodes were turned down. Through the extra DHash layer, the CFS service network achieves better robustness than the pure Chord network through cached and replicated copies of a data block [17]. The CFS authors published the evaluation results on the effect of node failures in an experiment consisting of a set of trials with different fractions of failed CFS nodes [17].

We re-conducted this experiment through Weevil according to the original experiment setup [17]. We considered a CFS service network with 1,000 CFS nodes distributed on a 50-host Emulab testbed with every 20 CFS nodes running on each testbed host.

Through one of the CFS nodes, a single CFS user actor first inserted 1,000 files into the subject system. Then a fraction of the CFS nodes were turned down at the same time. Ten seconds later, the single CFS user actor began to issue retrieval requests. The 1,000 retrieval requests were issued in a constant interval for the randomly selected keys. In the different experiment trials, we changed the fraction of failed nodes from 0.05 to 0.5 with the testbed, the SUE, and the user actor kept the same. To implement the simultaneous fault injection, we associated an environment actor with each CFS node scheduled to turn down. All the environment actors shared the same behavior model, which simply issues one fault injection action at a pre-determined time. Thus, this simultaneuous fault injection scenario can be simply modeled with an environment workload and the actor's workload interpretation program. The environment workload has the same number of lines as that of the failed nodes in the trial. All the workload lines have the same timestamp. The workload interpretation program translates each workload line to the shell command calling the stop script of the associated CFS node. For each experiment trial, we simply adjusted the number of environment actors in the trial configuration file.

Through experiment setup, Weevil was able to partition the unified environment workload to a number of one-line per-actor workloads and adjusted the trial master script. We collected six metrics for Weevil to configure, setup and deploy the experiment trials as shown in Table 8.1. **Configuration lines** is the total number of lines in our configuration inputs to Weevil for each trial, including the trial configuration file, the user actor programs, and the environment actor programs. **Configuration differences** is the number of lines of the trial configuration file that are changed between the successive trials. **Setup time** is the time it takes Weevil to perform the trial setup phase, i.e., the time to perform the workload correlating (action 1 in Figure 5.1), workload partitioning (action 2 in Figure 5.1), and script generation (action 3 in Figure 5.1). **Generated files** is the number of files created by Weevil for each trial. **Script differ-**

**ences** is the number of different lines in the generated scripts between successive trials. **Deployment time** is the time it takes Weevil to perform the trial deployment phase, i.e., the time it takes Weevil to deploy the generated trial control system to the testbed, plus the time it takes to start the SUE components and trial monitoring daemons on all the testbed hosts and to synchronize all the actors (action 3 in Figure 5.1). The CFS binary distribution has been pre-deployed on the testbed, thus is not considered in the deployment time. In each column of the table for the configuration and script differences, the value given is relative to that of the trial represented in the previous column. Since this is a measure of difference, it is not applicable to the first experiment trial in the table. We separate this table into two parts: all the metrics in the upper part are the costs for Weevil to conduct a trial, while those in the lower part show the processing complexity of Weevil.

| **Num of env actors** | **50** | **100** | **200** | **300** | **400** | **500** |
|---|---|---|---|---|---|---|
| **Config. lines** | 90 | 90 | 90 | 90 | 90 | 90 |
| **Config. differences** | – | 3 | 3 | 3 | 3 | 3 |
| **Avg. setup time (sec)** | 176 | 234 | 436 | 747 | 1156 | 1648 |
| **Avg. deploy. time (sec)** | 64 | 65 | 74 | 78 | 80 | 88 |
| **Generated files** | 2207 | 2307 | 2507 | 2707 | 2907 | 3107 |
| **Script differences** | – | 1900 | 3800 | 3800 | 3800 | 3800 |

Table 8.1: Environment Actor Modification

The 90 lines of configuration inputs are made up of the 37 lines in the trial configuration file, the 36 lines in the user workload interpretation program, and the 17 lines in the environment workload interpretation program. Between the experiment trials, only three lines in the configuration file need to be changed to tune the number of environment actors. Given these inputs, Weevil generated a customized trial control system consisting of a large number of trial scripts, programs, and auxiliary files. The setup time increases roughly linearly with the number of the environment actors since each environment actor has a customized actor program generated. Through parallel communication and file transfer, the overall deployment time shows good scalability.

From the table, we could see that through Weevil, the relatively minor costs resulted in a large number of effects.

For each trial of this experiment, we logged all the retrieves and their response times from the single user actor. The log was analyzed to study the failed retrieves because of the key loss caused by the CFS node failure. Figure 8.4 shows the min, median, and max values of the fraction of all the retrieval requests that failed in each trial. They are plotted as a function of the fraction of the failed nodes.

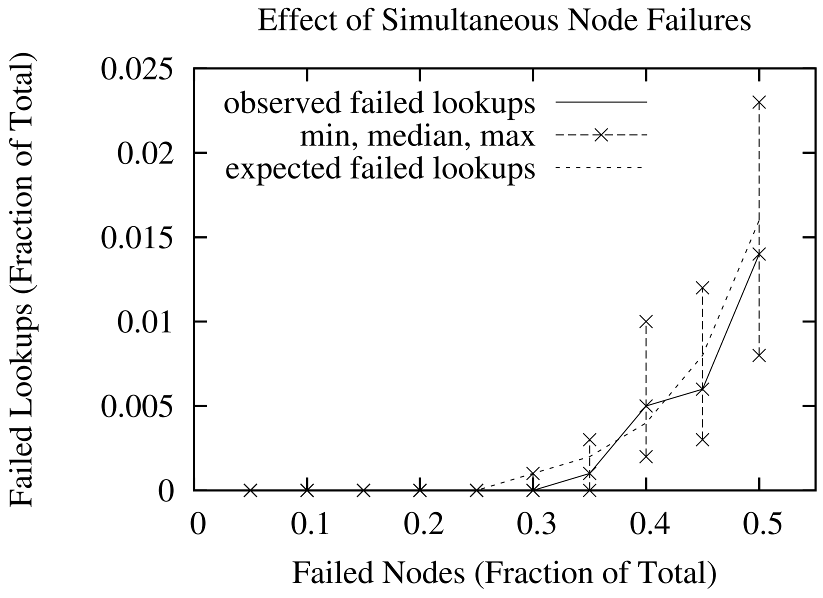


Figure 8.4: The Fraction of Failed Retrievals as a Function of the Fraction of the Failed CFS Nodes. The data set for each node failure fraction were observed in 5 experiment trials involving 1,000 block lookups; the bars indicate the minimum, median, and maximum of each data set.

The other plot in Figure 8.4 shows the corresponding theoretical values calculated based on the CFS block replication strategy. We used the default CFS service configuration of six replicas for each data block. Thus retrieval failures happened only when all the six replicas of a data block were lost because of the failure of enough CFS nodes. Assuming the fraction of failed CFS nodes is F, the probability of losing all of a data block's replicas is $F^6$ [17]. Based on this formula, we calculated the expected fraction of failed lookups as shown in Figure 8.4. The data observed in our five experiment trials for each fraction value are consistent with the expected value.

The Chord authors further complement the block replication strategy with the erasure-coded fragment strategy to reduce bandwidth consumption [18]. By default, the erasure-coded fragment strategy creates 14 (**efrags**) erasure-coded fragments of each block, 7 (**dfrags**) out of which are necessary to reconstruct the block. Whereas, if a block's size is less than 1210 bytes (**MTU**), the strategy calculates the optimal value of **dfrags** with the formula $(len + (MTU - 1))/MTU$ and replaces 7 with this optimal number. Our experiments stored files of 200 bytes in the CFS network. Thus, one $((200 + 1209)/1210)$ fragment is sufficient to reconstruct a file. It is as though the CFS network kept 14 replicas of each file. The probability of losing all of a file's replicas when half of the CFS nodes is down is $0.5^{14} = 0.000061$, which can be neglected. For the experiments with lower fractions of failed nodes, the error rate is even lower.

We reconfigured the CFS network to use this strategy. Then we re-conducted the above experiment with the same series of node failure fractions. This time, we did not observe any retrieval request that failed because of the loss of stored blocks, while there were some errors that happened within a short time after the node failure because of the inconsistent routing information on the CFS nodes. Some other successful retrieval requests took a long time to get responses because of the loss of some replicas of a data block. However, we observed that on average, the retrieval requests sent out 550 seconds after the node failure did not delay any more since the CFS service network had already recovered.

**Continuous Fault Injection**

Because of its distributed nature, node failure of a distributed system normally happens from time to time instead of once with many nodes failing at the same time. Thus, its performance under such continuous fault injection scenario is more interesting.

From the above experiment, we learned that the CFS service network needs some time to recover after node failure. After the network recovers, the routing table on each CFS node gets fixed and the lost data block replicas get re-built. But during this recovery process, the retrieval errors and delays may happen because of the inconsistency and loss of data block.

Thus, we were interested to know how the CFS service network performs under such continuously inconsistent and block-missing states. The Chord network underlying the CFS service network has been evaluated through a simulation in which randomly selected nodes continuously join and leave the simulated Chord network [54]. We studied a CFS network under the same scenario. Instead of simulation, we conducted another real experiment with the CFS network on an Emulab testbed.

We considered a 100-node CFS network distributed on a 50-host Emulab testbed with 2,000 files already inserted. Each CFS node has a user actor constantly issuing 50 retrieval requests for uniformly random file keys. The testbed, the SUE, and the user actors were kept the same across the experiment trials.

We changed the mean of the CFS node failure and joining rate R from 0.01 to 0.1 across the experment trials. These changes were reflected in the different environment workloads and the corresponding environment-actor mapping configurations. We used the analytical workload generator contained in the CFS binary distribution[2] to generate an environment workload for each trial that contains the continuous node failure and joining events with the mean event arrival rate equal to R. The events to every different CFS node in the workload are performed by a different environment actor. In each trial configuration, the associations between the environment actors and the CFS nodes are specified. Each environment actor thus has a list of actions to start up and bring down its associated CFS node based on the events in the workload. This continuous fault injection scenario thus is simply modeled with this environment workload and the actor's workload interpretation program to translate these events to the real actions.

[2] http://pdos.csail.mit.edu/chord

We measured the fraction of failed retrievals caused by the CFS network inconsistency in these experiments. Figure 8.5 shows the results as a function of the rate (over time) at which nodes fail and join.

Because of the DHash layer, we observed much smaller fraction of failed retrievals compared to the simulation results observed for the pure Chord network [54]. We further analyze the results of this experiment together with the next experiment.

**Reactive Fault Injection**

The above experiment based on a statistical fault injection model reflects an abstract failure scenario. Such a model regards node failure and recovery as random accidents in the system execution environment. However, for a distributed system deployed for real use, the exceptions are frequently caused by the incorrect operation or the heavy load on the distributed system. We further used Weevil to conduct an experiment with the CFS service network under such realistic failure scenarios.

In real setup of a distributed system, the system components with heavier load have higher probability to fail than the others. This experiment is to model such a fault injection scenario that is reactive to the real system execution status. Our experimental goal is to testify that the failure of the heavier-loaded CFS nodes influences more on the system performance than that of the randomly selected CFS nodes.

We considered a 100-node CFS system distributed on a 50-host Emulab testbed with 2,000 files already inserted. Each CFS node had a user actor constantly issuing 50 retrieval requests for randomly chosen file keys. We reused the user workload generated for the previous continuous fault injection experiment.

Since we cannot decide which CFS node to fail before trial execution, we associated an environment actor with each CFS node. Each actor's portion of the environment workload is a list of conditional "STARTUP" and "BRINGDOWN" actions in a constant interval. Whether they would be issued or not depends on the two variables of the associated CFS node, the `rpcrate` and the `liveornot`. If the node's remote pro-

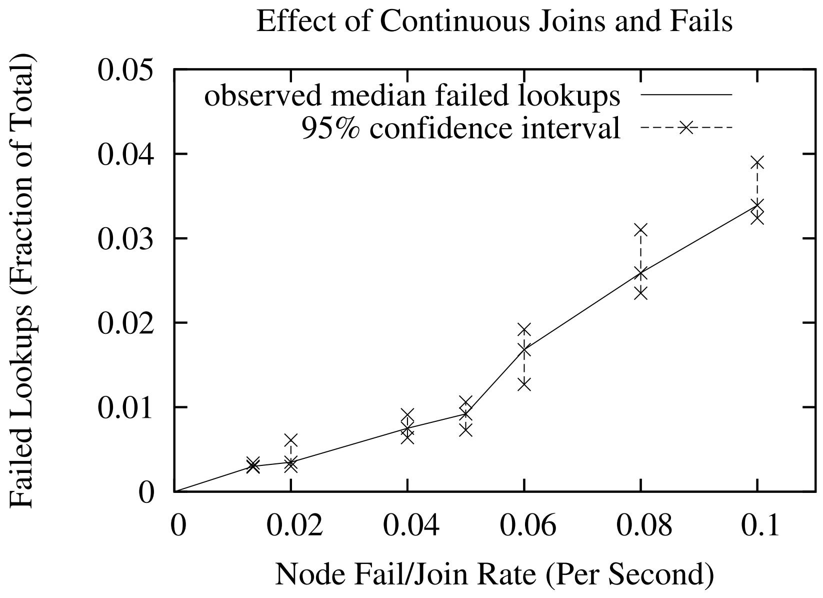


Figure 8.5: The Fraction of Failed Lookups as a Function of the Rate (over Time) at Which CFS Nodes Fail and Join. Only failures caused by the routing table inconsistency are included, not failures due to loss of all the replicas for a block. The data points for each node failure fraction were observed in 5 experiment trials involving 5,000 block lookups; the bars indicate the 5th, the median, and the 95th percentile of each data set.

cedure call (RPC) communication rate `rpcrate` is higher than a specified threshold, the scheduled "BRINGDOWN" action is issued. If the node is down upon a scheduled "STARTUP" action, i.e., `liveornot` is equal to 0, the "STARTUP" action is issued. In the environment workload, a conditional "BRINGDOWN" action appears every 10 seconds and a "STARTUP" action every 120 seconds. For example, the following snippet for actor A0 (represented in the workload generation as the workload process U0) is extracted from the generated environment workload when we assign the threshold value to 30:

```
event(10,U0,BRINGDOWN(rpcrate<30,rpcrate)
event(20,U0,BRINGDOWN(rpcrate<30,rpcrate)
...
event(120,U0,STARTUP(liveornot==0,liveornot)
...
```

The workload interpretation program implements the translation from the conditional actions to the real requests. The dependency modeled in the workload interpretation program includes two parts. First, we manually instrumented the CFS source code to log the RPC activities of the CFS nodes. Each environment actor accesses the CFS node's log for the values of `rpcrate` and `liveornot`. Based on their values, the actor conditionally executes the SUE component start/stop scripts. Second, since the CFS node's start script requires a parameter to specify which bootstrap node to contact, each environment actor needs to know at least one live node in the CFS network to bootstrap.

Figure 8.6 shows the sequence diagram of an example scenario with three environment actors. As shown in the figure, the environment actors in this experiment can be divided into two types, the **bootstrap actor** that is associated to the Chord network's initial bootstrap node (B0), and all the other actors (B1 and B2). Both types share the same actor programs. Their only difference is that the former type originally keeps a file listing all the live Chord nodes (C0, C1, C2). The following environment

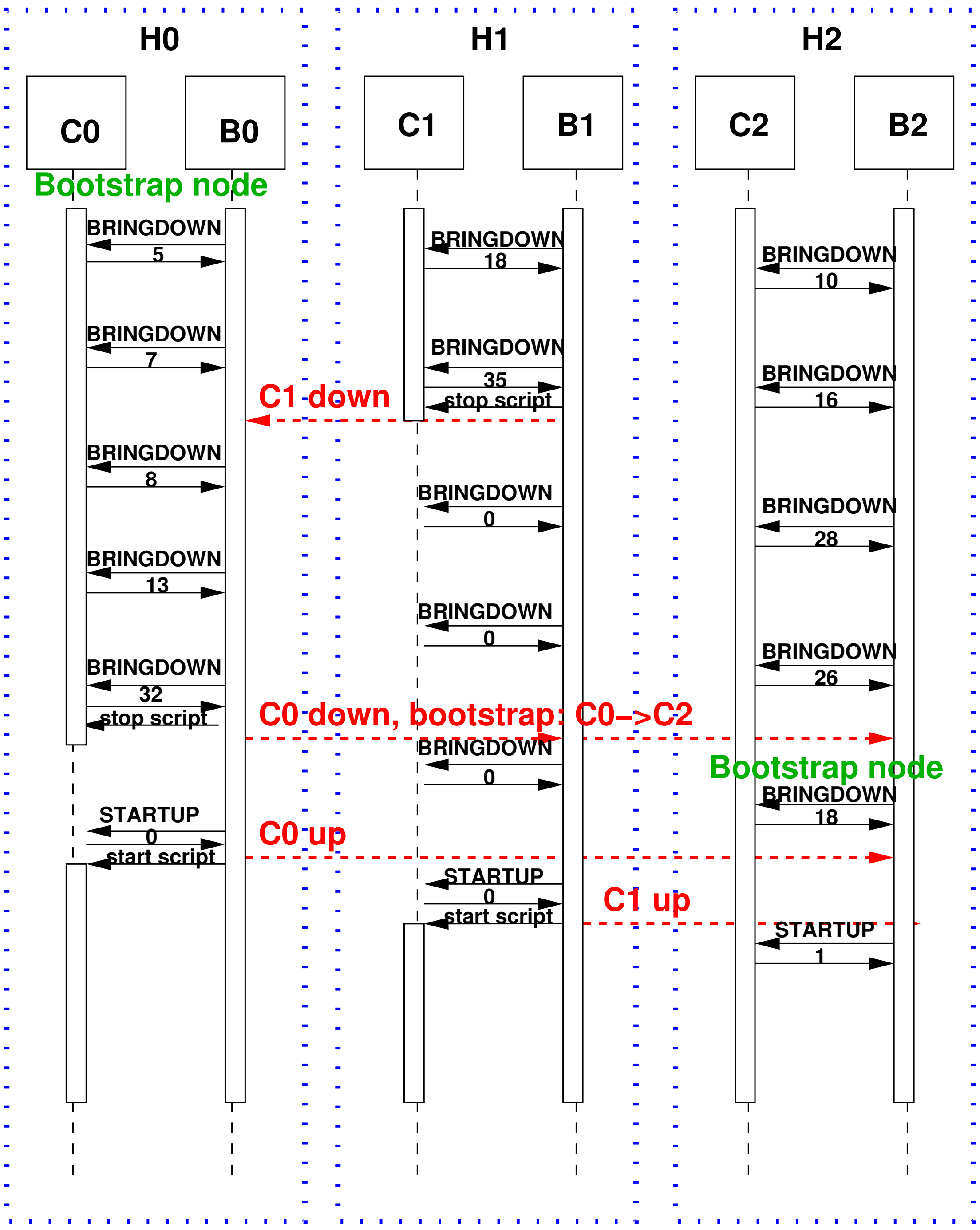


Figure 8.6: Sequence Diagram of Three Typical Reactive Environment Actors

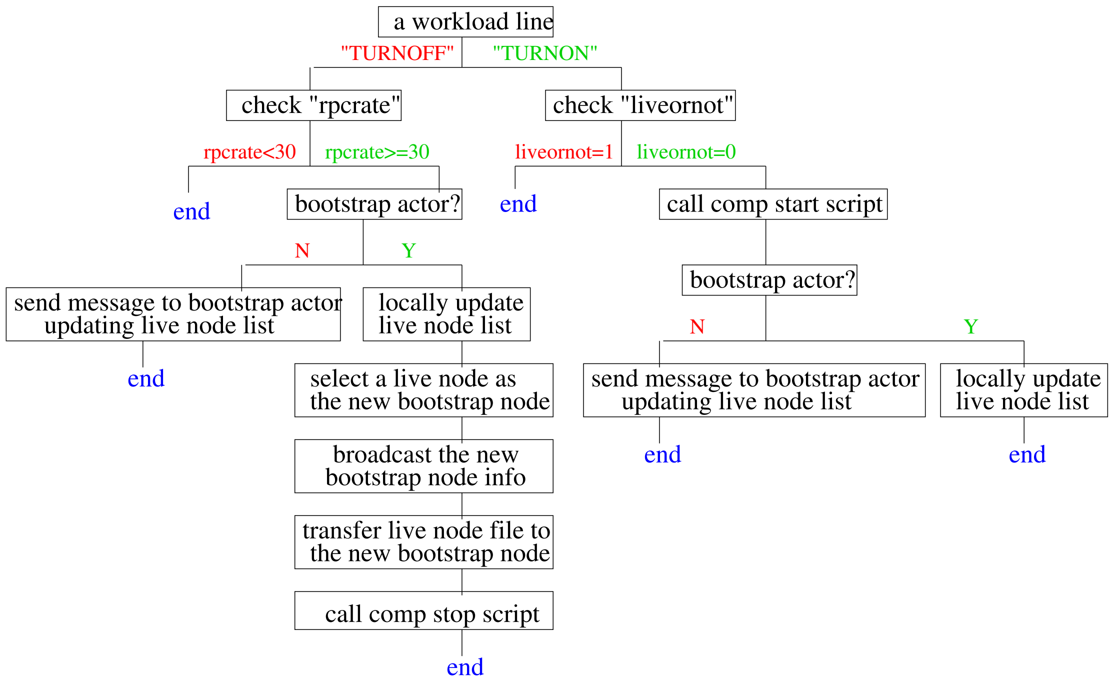
actor configuration snippet shows that the two types of actors are only different in the parameter `binaryDistDir` since the directory `bootactor` also includes the file listing all the live Chord nodes.

```
WVL_SYS_ActorType('InjProgram0', 'Shell', './actor', 'injector.sh',,, 'receiver.sh')
WVL_SYS_ActorType('InjProgram1', 'Shell', './bootactor', 'injector.sh',,, 'receiver.sh')
WVL_SYS_Actor('B0', 'I0', 'InjProgram1', 'C0')
WVL_SYS_Actor('B1', 'I1', 'InjProgram0', 'C1')
WVL_SYS_Actor('B2', 'I2', 'InjProgram0', 'C2')
```

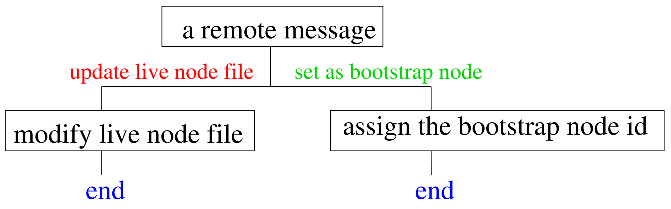
As shown in Figure 8.6, each actor constantly checks for the conditions of the "BRINGDOWN" and "STARTUP" actions. For example, the workload interpretation program "injector.sh" translates a "BRINGDOWN" action in this way: it first checks the RPC rate; if the returned rate is less than 30, nothing happens; otherwise, the stop/start script of its associated Chord node is called to turn off or turn on the node. Whenever this happens, the actor also passes a message to the bootstrap actor notifying it that a node is down (shown as the dotted arrow in Figure 8.6). Upon receiving the message, the bootstrap actor updates its list of live nodes.

If the bootstrap node is turned down, the situation is more complicated. Since the original bootstrap node is down, another live node must take over the role (such as the replacement of C0 with C2 in Figure 8.6 when C0 is turned down.) Along with the replacement, the file listing all the live nodes that C0 kept before is transferred to C2 and all the other actors will be notified of the replacement. Thus all the messages updating the list of the live nodes will be passed to the new bootstrap actor afterwards.

We implemented the above actor behavior model through a 90-line workload interpretation shell script and a 45-line message receiving program. The flow charts of these programs are shown in Figure 8.2.3. We conducted a set of trials with the value of the RPC rate threshold equal to 13, 14, 15 and 16 respectively. The different threshold value caused different node failure rates over time. We performed five trials for each threshold value, thus totally 20 trials in this experiment. Each trial lasted 5000 seconds.


a workload line
"TURNOFF"
"TURNON"
check "rpcrate"
check "liveornot"
rpcrate<30
rpcrate>=30
liveornot=1
liveornot=0
end
bootstrap actor?
end
call comp start script
N
Y
send message to bootstrap actor updating live node list
locally update live node list
bootstrap actor?
end
select a live node as the new bootstrap node
N
Y
send message to bootstrap actor updating live node list
locally update live node list
broadcast the new bootstrap node info
end
end
transfer live node file to the new bootstrap node
call comp stop script
end

(a) Workload Interpretation Program



(b) Message Receiving Program

Figure 8.7: Flow Charts of the Actor Programs for the Reactive Fault Injectors

We separated the log file of each trial into five segments, each of which is 1000-second long. For each segment, we calculated the median of the retrieval failure rate versus the node failure rate over time. We plotted the results in Figure 8.8 in comparison to the median plot from the previous experiment.

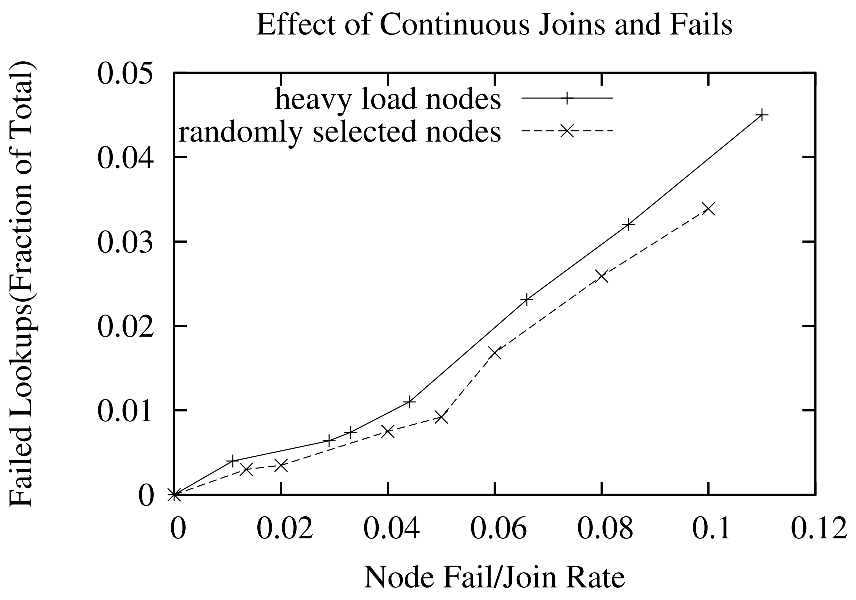


Figure 8.8: Median of the Fraction of Lookups that Fail as a Function of the Rate (over Time) at Which Nodes Fail and Join. Only failures caused by the routing table inconsistency are included, not failures due to loss of all the replicas for a block.

As shown in the figure, although the reactive fault injection scenario is just a special case of the continuous fault injection, we observed higher retrieve failure rates in this experiment than the corresponding results from the last experiment, in which we randomly chose nodes to fail and join. It validates our prediction that the failure of heavier loaded nodes has more effect on the performance of the CFS network.

Different from the fault injection model in either of the previous experiments, which is pre-determined before trial execution and can be fully described in an environment workload, the model in this experiment is dynamicly dependent on the SUE execution status. Thus, the level 4 actor behaviors need to be modeled to capture the reactive fault injection scenario. As indicated in Table 4.1, we implemented such behaviors with a workload containing conditional workload lines, and the actor programs with the ability to translate the workload lines and the ability to analyze the trial-execution-time information.

**Malicious Fault Injection**

In Chapter 6, we discussed that the workload correlating action (action 1 in Figure 5.1) in our trial automation framework is able to adjust the environment workload and the user workload to model the special scenarios that require the specific combination of a sequence of user behaviors and environment variations. In this experiment, we examplify the usage of this action by modeling a malicious fault injection scenario to study the CFS service network.

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 as shown in Figure 8.9. Each CFS node has its logical successor in the ring to be the first entry in its finger table. Thus, upon receiving a retrieve request, a CFS node always first tries to contact its successor. If the successor is unreachable, it tries to contact the next entry in its finger table.

In this experiment, we use Weevil to evaluate the performance of a CFS network under a malicious fault injection scenario. In this scenario, the environment actors learn of the routing algorithm. They thus maliciously bring down some CFS nodes from time to time. Instead of randomly selecting nodes in the continuous fault injection scenario, each environment actor in this scenario always choose to bring down the successor of the CFS node that a future request will be sent to. In other words, the action is to attack the future request by bringing down the successor. We call the request "attacked request" in our discussion. Based on the CFS routing algorithm, we expect those attacked requests to have longer reponse time but not to fail if this scenario can be correctly modeled and applied in the experiment using Weevil.

We considered a 100-node CFS network distributed on a 50-host Emulab testbed

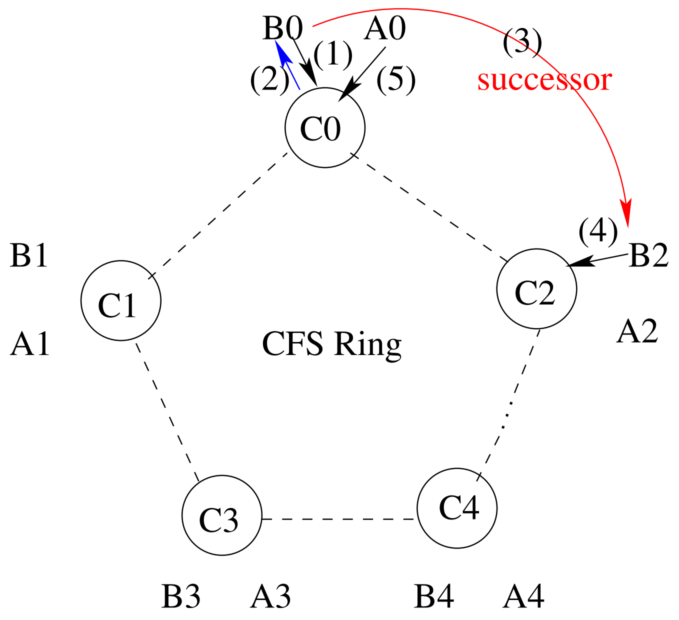


Figure 8.9: Malicious Fault Injection Scenario. In this scenario, A0 is about to issue a "get" request. The malicious attack was implemented to be the following sequence of actions: (1) B0 sends out "getsucc" request to C0 to get the successor identity of C0; (2) The successor identity (C2) is returned to B0; (3) B0 communicates with the environment actor B2 associated to C2 to activate a BRINGDOWN action; (4) B2 brings down C2; (5) A0 issues "get" request to retrieve a file.

with 2,000 files already inserted. Each CFS node has a user actor constantly issuing 50 retrieval requests for uniformly random file keys every 100 seconds. We reused the user workload and the environment workload generated in the continuous fault injection experiment. These two workloads are originally independent. The actions contained in the environment workload can be configured to be performed by any environment actors in an experiment trial. However, in this malicious scenario, we adjusted the environment workload to always bringing down the successor to the node that is going to receive retrieval request from the user workload. Figure 8.10 examplifies the inputs of the two types of workloads and the output adjusted environment workload.

The original environment workload schedules the “BRINGDOWN” and “STARTUP” actions at the timestamp 242 and 439. We intend each “BRINGDOWN” action to attack a future request, in this example, the request sent 5 seconds later. The 5-second period is saved for a CFS node to really go down. Thus, the “BRINGDOWN” action at 242 should attack the retrieve request `event(247,U48,GET(f1900))` in the user workload. Since U48 is mapped to I48 in the trial configuration file, the environment workload line at 242 is adjusted to `event(242,I48,BRINGDOWN())`.

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 bringing down or starting up its associated CFS node upon a message notifying that its associated node is the successor to access for a future attacked request. The successor message comes from the environment actor that is going to send out the attacked request. The actor gets the successor identity using a Chord client program “getsucc.” This actor behavior model includes level 4 actor behaviors, i.e., whether or not an actor should turn off or turn on a CFS node is dependent on the trial-execution-time response to the **getsucc** program and the inter-actor communication message.

As expected, we observed no failure for the attacked requests. A few failed requests were caused by the CFS network inconsistency. We divided all the retrieves into

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

(a)input user workload

```
event(242,I0,BRINGDOWN())
event(439,I1,BRINGDOWN())
...
```

(b)input environment workload

```
‘U’i <-> ‘N’i
‘I’i <-> ‘N’i
i=0..99
```

(c)trial configuration directives

```
event(242,I48,FAILSUCC())
event(439,I12,FAILSUCC())
...
```

(d)adjusted environment workload

Figure 8.10: Workload Adjustment to Correlate the User Workload with the Environment Workload. The workload in (d) is derived from the workloads in (a), (b) and (c).

three types, the attacked requests, the next requests to the attacked sent by the same actor, and all the 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 experiment results are shown in Figure 8.11.

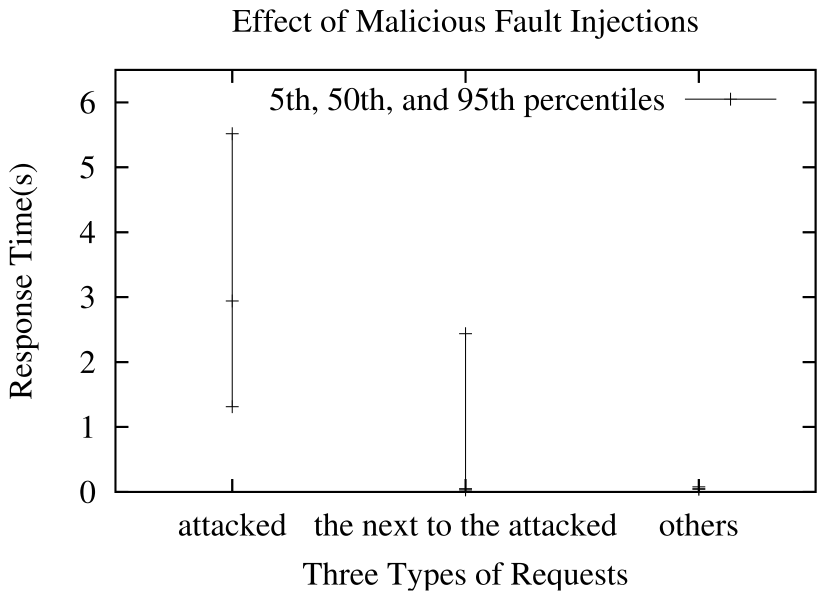


Figure 8.11: CFS Retrieve Responce Time under Malicious Attack

As shown in Figure 8.11, we measured the 5th, 50th, and the 95th percentile of the retrieve response time for the three types of requests separately. The attacked requests show longer retrieve latency than those unattacked ones because of the loss of the successors. But the effect does not last. The CFS network was able to fix the successor information in the routing table and the finger table so that the request to the same attacked node after 100 seconds could get the response quickly even if the original successor had not been turned on. These experiment results validated the functionality of our workload correlating activity in helping model experiment scenarios.

## 8.3 Scalability of the Framework

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

**Freenet Experiment**

Freenet has been undergoing a redesign of its core architecture: the "Next Generation (NG) Routing Algorithm" is intended to replace its "Classic Algorithm". According to Clarke's analysis of the two algorithms [14], the NG routing algorithm makes Freenet nodes much smarter about deciding where to route information when a new request is received. Based on that analysis, the NG routing algorithm should exhibit better scalability and show improved performance for most requests.

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

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

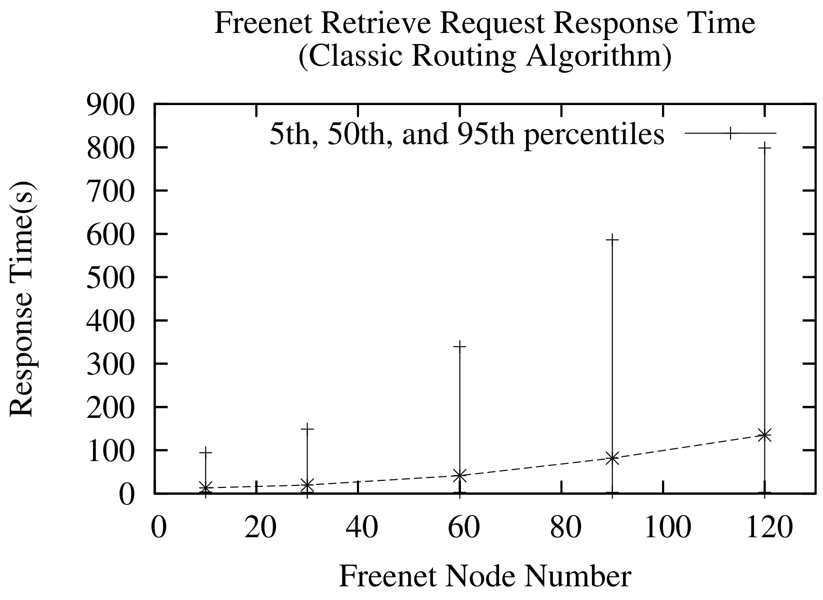
We applied the PlanetLab testbed model from Section 8.2.1 and the SUE model of Freenet from Section 8.2.2 to this experiment. Their only configuration differences between trials are the number of testbed host declarations and component declarations. However, the number of components, testbed hosts, and user actors are all equal for each of these trials since the **Component/Host** mapping and the **WorkloadProcess/Component** mapping are both one-to-one mapping. The workload entries of these trials are either insert requests or retrive requests. Thus, the actor programs were all implemented as a shell script that hands off the execution of each action in the workload to one of the following java commands that the Freenet binary distribution provides:

```
<javaPath> -cp <softwarePath>/freenet.jar freenet.client.cli.Main
     put <KEY> <filename>
<javaPath> -cp <softwarePath>/freenet.jar freenet.client.cli.Main
     get <KEY> <filename>
```

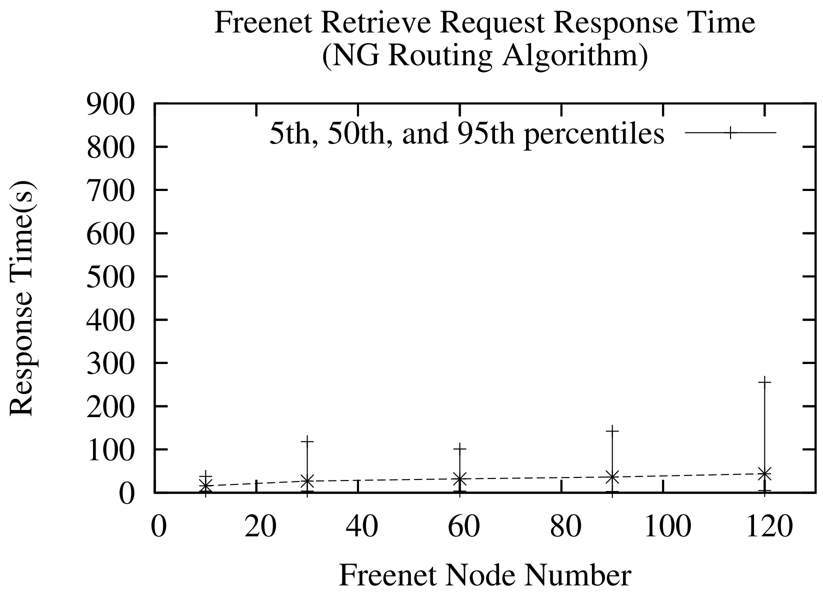
The unsolved parts in the above commands were all provided as macros in the parameter **argument** of the entity **ActorType** to serve as arguments to the shell script.

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

Figures 8.12a and 8.12b show the median, the 5th, and the 95th percentile of retrieve latency for the classic and NG routing algorithms, respectively. The median latency increases from 13 to 135s for the classic algorithm, and from 16 to 44s for the NG algorithm. Obviously, the classic algorithm has a significantly higher retrieve latency rate on average. This validates our expectations of the relative scalability of the two routing algorithms. Additionally, Figure 8.12 shows that the classic algorithm exhibits a larger standard deviation than the NG algorithm, indicating that the classic algorithm's



(a)



(b)

Figure 8.12: Freenet Retrieve Latency

performance fluctuates greatly for different requests, something which results from its ignorance of the underlying network topology. Another interesting observation is that all the trials have similar low 5th percentile latencies. These are caused by retrieval requests for files already cached locally on the physical site, and for files cached on a fast host and found after very few routing attempts. This indicates that the classic algorithm and the NG algorithm perform equivalently for these kinds of requests.

### Chord Experiment

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

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

A feature of Chord is that each lookup of a file is performed using the hashed key it returns when the file was inserted into the Chord network. Unfortunately, this complicates experimentation, since Weevil cannot adjust an experiment trial dynamically to data provided by the SUE. We can either solve this problem by pre-computing the hashed keys of the inserted files outside the Chord system before the trial execution or by performing two consecutive trials, one to examine insert latency, and the other to evaluate retrieval, with the keys and database created during the first trial being used during the next to keep all the inserted files. Although we were able to pre-compute the key that the Chord system returns for each inserted file, we decided to take the second approach to exercise Weevil in case the key is not easily computable before run-time for other distributed systems.

The model configurations for the two trials are the same except the database parameter we mentioned in Section 8.2.2. In the first trial, a database was created for each component in its **theCompPath**. Thus, we included `theCompPath/db` as the database parameter in the start script of component type **ChordNode**. To continue using it, in the second trial, we set the database parameter to a system-specific component property **DBPath** which refers to the database created in the first trial defined as follows in which `chord_a` is the experiment ID of the first trial.

```
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_ComponentProp(i, ‘DBPath’,
                theWeevilRoot‘/chord_a/ChordNode/’i)’, Nodes)dnl
```

We used the same workloads as those in the Freenet experiment, while the actions in the workloads are translated to one of the following Chord commands instead in Chord actor implementation:

```
<theSoftwarepath>/filestore <theCompPath>/sock -s <filename>
<theSoftwarepath>/filestore <theCompPath>/sock -f <KEY>
```

`filestore` is a Chord client program to insert or retrieve files to the Chord network indicated by `-s` or `-f` option. The Chord experiment results are shown in Figure 8.13.

The Figure shows the median, the 5th, and the 95th percentile of retrieve latency for Chord. The retrieve latencies are an order of magnitude less for Chord than for

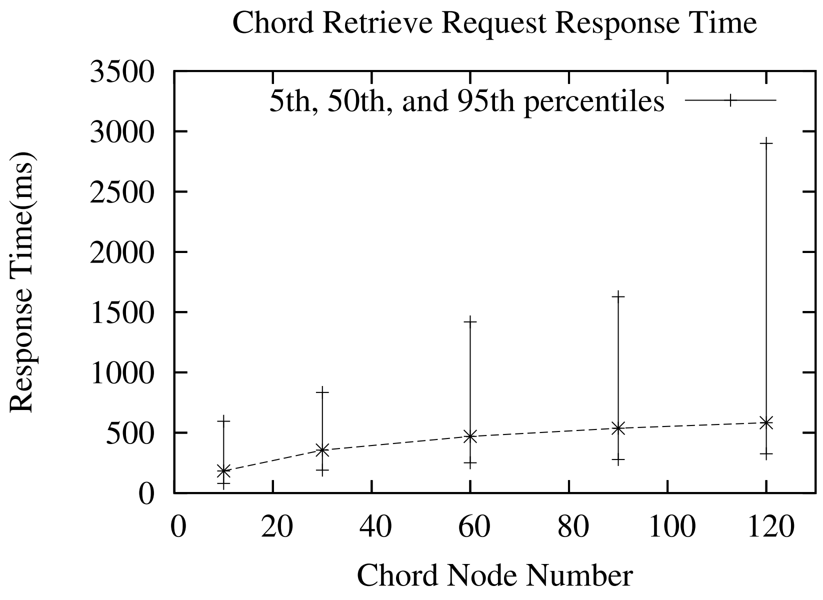


Figure 8.13: Chord Retrieve Latency

Freenet. For Chord, the median latency increases from 183 to 583ms. This is more pronounced than that reported by Chord's authors [54]. We believe this is due to a difference in trial configuration. In particular, although the Chord authors increased the number of Chord instances in their experiment trials, their testbed consisted of only ten hosts; experiment trials with more than 10 instances were conducted by running multiple independent copies of the Chord software at each site. In our experiment, we increased the geographical scale of the testbed as well as its node count. In that sense, our experiment is more realistic than that of Chord's authors, and the results likely to be more reliable.

**Discussion**

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

In Table 8.1, we quantitatively evaluated Weevil's performance in automating the experiment trials with different numbers of environment actors. We continue with such quantitative evaluation using Weevil in our Chord experiment on PlanetLab. As shown in Table 8.2, we collected five out of the six metrics we used in Table 8.1.

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

Table 8.2: Experiment Scale Modification

These numbers clearly illustrate that Weevil allows the experiment to be easily and quickly reconfigured to handle changes, that generation and automation are effective ways to configure and manage large scale experiments, and through parallel communication and file transfer, the overall deployment time scales well. Between these experiments, we made changes simultaneously in several dimensions, including the number of hosts, components, and user actors. A more systematic evaluation of Weevil's ability to leverage automation, with changes made in each dimension independently, is discussed below in Section 8.4.

## 8.4 Utility of the Automation Features

We now present more quantitative evaluation of the benefits of Weevil's automation features. In particular, we measured the effort involved in switching between different experiment trials with Chord performed on PlanetLab. We considered varying degrees of modification, from minor tweaks to substantial changes. Specifically, we considered testbed modifications, system parameter modifications, SUE modifications, and user actor modifications. We collected the same five metrics as those in Table 8.2 for each series of these experiments.

**Testbed Modification**

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

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

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

This mapping configuration evenly spreads the Chord nodes on the limited number of testbed hosts, with 12 Chord nodes per host on the 10-host testbed and 4 Chord nodes per host on the 30-host testbed. The Chord nodes on the same host cannot use the same listen port. As a result, the listen ports of the Chord nodes were updated through the following configuration changes:

```
< WVL_SYS_Foreach(‘i’, ‘WVL_SYS_ComponentProp(‘N’i, ‘ListenPort’, eval(27882+i/10))’,
< WVL_SYS_Range(‘’, 0, 119))dnl
----
> WVL_SYS_Foreach(‘i’, ‘WVL_SYS_ComponentProp(‘N’i, ‘ListenPort’, eval(27882+i/30))’,
> WVL_SYS_Range(‘’, 0, 119))dnl
```

With these simple configuration changes, Weevil was not only able to automatically deploy the system onto the larger testbeds, but also automatically relocate each actor to its corresponding component’s new host. The metric values of these trials are shown in Table 8.3.

The numbers in the third row of Table 8.3 highlight Weevil’s parallel file transfer capabilities, as the deployment time grows slowly with the number of hosts. The 10-host experiment has a longer deployment time than the 30- and 60-host ones because

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

Table 8.3: Testbed Modification

there is less opportunity for parallelism with so many components residing on each host. The most important result of this experiment is the amplification achieved by Weevil's generative capabilities. Small changes in the configuration were multiplied by a factor of up to 35 in the resulting generated scripts. This amplification, coupled with Weevil's consistently quick setup time, clearly demonstrates the benefits of automation and generation for the task of modifying the testbed. The number of files generated depends on the number of components and the number of actors, which were fixed for this experiment.

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

```
-v ‘WVL_Component_’WVL_Component_ID‘_vn’
```

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

```
> WVL_SYS_Foreach(‘i’, ‘‘WVL_SYS_ComponentProp(i, ‘vn’, 3)’, WVL_SYS_Range(‘’, 0, 119))dnl
```

The metric values of these trials are shown in Table 8.4. Once again, the data show how a few tweaks in the system configuration result in a large number of changes to the management scripts.

| **Parameter** | **Initial** | **Updated** | **New** |
|---|---|---|---|
| **Configuration differences** | – | 1 | 2 |
| **Setup time (sec)** | 114 | 114 | 114 |
| **Deployment time (sec)** | 59 | 59 | 59 |
| **Generated files** | 481 | 481 | 481 |
| **Script differences** | – | 120 | 120 |

Table 8.4: Parameter Modification

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

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

Table 8.5: SUE Modification

We altered the SUE of an existing experiment trial by increasing the number of components from 10 to 120, with the number of hosts and actors fixed at 120. To accomplish this we instantiated the new components and added their properties and relations. We also updated the SUE-to-testbed mapping to accommodate the new components. Again, all of these can be accomplished in just several lines utilizing the programmability of our trial automation framework. The metric values for these experiments are shown in Table 8.5. As the difference numbers show, even fundamental changes in an experiment can be accomplished with very small configuration changes, which are amplified by Weevil as necessary when generating files.

**User Actor Modification**

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

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

Table 8.6: User Actor Modification

The data for this experiment, shown in Table 8.6, confirm what the previous experiments showed, namely that our trial automation framework powered by the model-based generative approach 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.

## 8.5 Support of Unreliable Testbeds

As discussed in Chapter 7, we designed two mechanisms to guarantee the reliability of our trial automation framework in conducting experiment trials. In this experiment, we wish to measure the recovery time from a lasting error is detected until it is bypassed with the partial-redeployment mechanism.

We conducted an experiment with a 100-node Chord network on a 50-host Emulab testbed, each Chord node had a Chord user actor injecting retrieve requests. After the Chord user actors began processing their per-actor workloads, we manually logged onto an Emulab host and rebooted it with the shell command “sudo reboot”. Weevil was able to detect that unreachable host and thus began to re-constructed a revised trial to bypass the rebooted host.

We observed that the recovery time for Weevil to detect the error, terminate the experiment, generate the supplementary configuration file, and generate the updated trial scripts was 32 seconds. The deployment of the re-generated trial scripts took 21 seconds instead of 334 seconds if the whole trial control system and the Chord binary source code are all re-deployed. In addition, the whole process was automated. We only needed to provide one-line configuration for each backup host in the original trial configuration file, then we could leave the experiment trial running without the need to frequently check and fix errors.

# Chapter 9

# Conclusions

In this thesis we have presented a framework for automating experimentation with distributed systems on distributed testbeds. It is targeted at different types of highly distributed systems, removing many practical obstacles, such as scale and heterogeneity, that hinder their experimentation in real environments. We consider an experiment made up of a set of closely related trials. The automation framework works at the trial level. It automates the three key steps of each experiment trial: workload generation, trial deployment and execution, and trial post-processing.

We designed our approaches for the first two steps, namely the simulation-based workload generation approach and the model-based generative approach. The simulation-based approach offers a flexible, complementary workload generation means to the widely used analytical approaches. The model-based generative approach provides a higher-level automation service than other available experimental tools by automating the trial control system construction based on the configuration modeling of each experiment trial. In summary, through this thesis work, we made the following contributions to the research in promoting the experimentation activity of distributed systems:

(1) We proposed an approach to categorize distributed actor behaviors based on their dependency on other actor behaviors or on the real system execution.

(2) We designed a more flexible, scalable workload generation approach, the simulation-

based workload generation approach, which generates synthetic workloads for experiments with distributed systems by modeling the distributed actor behaviors using the discrete-event simulation programs.

(3) We created the experiment trial configuration models for analyzing and describing different aspects of experiments with distributed systems. Their configurations provide a generic and programmable way to construct the experimentation framework applicable to different distributed systems on various testbeds for a wide-ranging experimental goals.

(4) We provided a higher-level automation mechanism than other available experiment automation tools using generative techniques. The mechanism automatically translates the high-level trial model configurations into a customized trial control system, which can be directly used to automate the specific trial.

(5) We implemented a scalable experimentation framework that can be applied to wide-area network testbeds. Through the model-based generative techniques, the scale, heterogenity, and instability of the wide-area network testbeds are transparent to the experimenter. Parallel processing is used to largely improve the efficiency of the large-scale experiments.

(6) We designed the recovery and bypassing mechanisms to guarantee the reliability of the experiment results produced in the experimentation framework, which is especially valuable for those wide-area, public testbeds.

To date we have used the prototype Weevil on a broad set of distributed systems drawn from three common operating paradigms: event-based, peer-to-peer, and traditional client/server. Further, we have used Weevil to perform meaningful experiments on local-area, wide-area, and emulated testbeds. Weevil allowed us to create workloads for real scenarios and to successfully reproduce and broaden previously published experimental results. Finally, we have evaluated Weevil's ability to leverage automation across multiple experiment trials and to quickly recover from network failures. All of these support our confidence in the feasibility and the advantages of the above research results.

However, this thesis work is still an attempt to explore new approaches to improve the experimentation activities in engineering distributed systems. The research in the experiment automation area is far from completion. This thesis work is still limited in its applicability and its level of automation. These limitations indicate some interesting research directions to explore in the future. In the following, we discuss the limitations of this thesis work and the possible future research topics to solve those limitations.

**Experiment-Level Improvement**

This thesis work is focused on each experiment trial, in which the SUE's parameter setting, its execution environment and its execution scenario have been decided. The trial process is repeated for many times in an experiment with different configurations. Thus, its automation and the easy tuning between trials supported by our trial automation framework can greatly promote the experimentation activity with distributed systems. However, other than the large-scale, heterogeneity and instability, the challenges in experimenting with a distributed system also include its large space of possible configurations, which is not covered by this thesis work.

Two main research questions arise from this challenge. One question is on experimentation design, i.e., how to choose a managable and typical subset of all the possible combinations of the system parameter values. The other is on experiment-level automation, i.e., how to improve the efficiency in conducting a set of trials. Both of these two questions have been addressed in the available research work [44, 65, 37]. However, many problems still remain to be done in the future research. One limitation of the current work is the lack of automation in the solutions. Much human intervention is required. Also the applicability of the approaches have not been thoroughly studied.

To our knowledge, the current techniques have only been applied to QoS evaluation of distributed real-time and embedded systems. In addition, the scalability, cost-efficiency and reliability of the techniques have not been well studied.

**Distributed System Testing**

Our previous experiences with the experimentation framework are all to study the performance of the SUE. However, as one type of experimentation, the distributed system testing can also get help from the framework with some extra supports. One of the supports is the replacement of synthetic workload generation with test suite generation for distributed systems. Besides, distributed systems are inherently nondeterministic. Two executions of the same system may produce different, but nevertheless valid results. It is difficult to reproduce errors and to judge the correctness of some unexpected situations. Therefore, applying the experimentation framework to distributed system testing is a difficult but important research topic.

**Client Behavior Study and Failure Scenario Modeling**

Our simulation-based workload generation approach provides a means for the software engineers to generate workload based on their intuitive feelings about the distributed system execution scenarios, which are limited and sometimes misleading. As we have pointed out, an optimal workload should be able to model some type of realistic scenarios. Thus, systematic studies on the distributed client behavior and on the network variation scenarios are necessary to make the simulation-based workload generation more useful and dependable. Unfortunately, there have been very few endeavors in this direction.

**Automation Support of Actor Behavior Modeling**

In our framework, the actor behaviors are modeled with the combination of a textual workload capturing the determined aspects and the actor programs implementing the dynamic aspects. Such design considers the dynamism of the distributed actor behavior that is often neglected by the available workload generators or test generators.

However, the separation of determined parts from the dynamic parts also make our analysis of actor behaviors more complicated. The non-standard programming of the trial-execution-time inter-actor communication can also cause unexpected distortion to the experimental results. A research topic arising from this problem is the automation support of actor behavior modeling. A mechanism through which an integrate description of the actor behavior can be automatically divided into a determined part and a lightweight and efficient dynamic part will be very useful.

**Distributed Data Analysis**

Our trial automation framework covers three steps of each trial, while we take a very simple approach for the last step, trial post-processing. Our approach is enough for collecting and analyzing data in the experiments we conducted since we mainly measured the system's overall performance seen by system clients. However, the real data generated by distributed systems can be very complicated. Events happening at different locations of a distributed testbed can be correlated. If a software engineer wishes to find out the problem source of an interesting phenomenon observed in a distributed system, he or she needs to study the large quantities of events and states of the system components. Distributed data analysis becomes necessary to handle these data and extract useful information. Thus, distributed data analysis is another research direction to explore to make the trial automation framework more useful. Some remaining research questions in this direction include how to correlate distributed data as well as the workload actions causing the data, how to make the on-the-fly data analysis less intrusive to the trial execution, and whether there is a generic way to derive the tasks and rules of distributed data analysis based on the model of the SUE.

**Incremental Recovery**

In our current framework reliability mechanisms, we take the simple approach to regard the trial execution phase as an individual work unit and always re-conduct it as a whole if network failure happens. Such an approach prevents the distortion of the trial results caused by the network failure. However, it makes the recovery process less efficient. The successfully finished actions in the workloads need to be re-performed. Thus, the incremental recovery of the trial execution phase would be very helpful to improve the recovery process of a failed trial.

# Bibliography

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

[2] Ehab Al-Shaer, Hussein Abdel-Wahab, and Kurt Maly. Hierarchical filtering-based monitoring system for large-scale distributed applications. In the 10th International Conference on Parallel and Distributed Computing Systems, New Orleans, Louisiana, October 1997.

[3] Saurabh Bagchi, Gautam Kar, and Joe Hellerstein. Dependency analysis in distributed systems using fault injection: Application to problem determination in an e-commerce environment. In the 12th International Workshop on Distributed Systems: Operations and Management (DSOM '2001), Nancy, France, October 2001.

[4] E.C. Bailey. Maximum RPM. Red Hat Software, Inc., February 1997.

[5] Anand Balachandran, Geoffrey M. Voelker, Paramvir Bahl, and P. Venkat Rangan. Characterizing user behavior and network performance in a public wireless lan. In SIGMETRICS, Marina Del Rey, CA, 2002. ACM.

[6] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In Sigmetrics, 1998.

[7] Mudashiru Busari and Carey Williamson. Prowgen: A synthetic workload generation tool for simulation evaluation of web proxy caches. In IEEE INFOCOM, 2001.

[8] Mauro Caporuscio, Antonio Carzaniga, and Alexander L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. IEEE Transactions on Software Engineering, 29(12):1059–1071, December 2003.

[9] Mark Carson and Darrin Santay. Nist net – a linux-based network emulation tool. ACM SIGCOMM Computer Communication Review, 33(3):111–126, July 2003.

[10] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems, 19(3):332–383, August 2001.

[11] Ramesh Chandra, Ryan M. Lefever, Michel Cukier, and Willian H. Sanders. Loki: A state-driven fault injector for distributed systems. In Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000), pages 237–242, New York, NY, June 2000.

[12] Brent Chun. pssh HOWTO. Intel Research Berkeley, November 2003.

[13] Brent N. Chun. Dart: Distributed automated regression testing for large-scale network applications. In Proceedings of the 8th International Conference on Principles of Distributed Systems, Grenoble, France, December 2004.

[14] Ian Clarke. Freenet's Next Generation Routing Protocol. Freenet Project, July 2003.

[15] Ian Clarke, Scott G. Miller, Theodore W. Hong, Oskar Sandberg, and Brandon Wiley. Protecting free expression online with Freenet. IEEE Internet Computing, 6(1):40–49, 2002.

[16] K. Czarnecki and U.W. Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[17] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01), pages 202–215, Chateau Lake Louise, Banff, Canada, October 2001.

[18] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a DHT for low latency and high throughput. In Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04), San Francisco, California, March 2004.

[19] Laurent Destailleur. AWStats Logfile Analyzer Documentation, December 2005.

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

[21] Alexander Egyed. Dynamic deployment of executing and simulating software components. In Proceedings of the Second International Working Conference of Component Deployment(CD 2004), pages 113–128, Edinburgh, UK, May 2004.

[22] Kevin Fall and Kannan Varadhan. The ns Manual. The VINT Project, June 2006.

[23] András Faragó. A graph theoretic model for complex network failure scenarios. In Proceedings of the Eighth INFORMS Telecommunications Conference, Dallas, Texas, March 2006.

[24] N.E. Fenton and S.L. Pfleeger. Software Metrics: A Rigorous and Practical Approach. PWS Publishing Company, 2nd edition, 1998.

[25] Patrick Goldsack, Julio Guijarro, Antonio Lain, Guillaume Mecheneau, Paul Murray, and Peter Toft. Smartfrog: Configuration and automatic ignition of distributed applications. In Proceedings of the 2003 HP Openview University Association conference(HP OVUA 2003), Geneva, Switzerland, July 2003.

[26] 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.

[27] David Guerrero. System administration. Linux J., 1999(58es):11, 1999.

[28] Krishna P. Gummadi, Richard J.Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In the Proccedings of the 19th ACM Symposium on Operating Systems Principles (SOSP-19), Bolton Landing, NY, October 2003. ACM.

[29] R.S. Hall, D.M. Heimbigner, A. van der Hoek, and A.L. Wolf. An Architecture for Post-Development Configuration Management in a Wide-Area Network. In Proceedings of the 1997 International Conference on Distributed Computing Systems, pages 269–278. IEEE Computer Society, May 1997.

[30] R.S. Hall, D.M. Heimbigner, and A.L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. In Proceedings of the 1999 International Conference on Software Engineering, pages 174–183. Association for Computer Machinery, May 1999.

[31] Helmut Hlavacs and Gabriele Kotsis. Modeling user behavior: A layered approach. In MASCOTS'99, IEEE Computer Society, 1999.

[32] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. IEEE Computer, 30(4):75–82, 1997.

[33] Alex Hubbard, C. Murray Woodside, and Cheryl 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.

[34] IEEE. IEEE Standard for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks, 2005.

[35] InstallShield Corporation. InstallShield, 1998.

[36] Karl Jeacle. Marimba's castanet. Broadcom Technical Journal, 3(1):44–45, March 1997.

[37] Arvind S. Krishna, Emre Turkay, Aniruddha Gokhale, and Douglas C. Schmidt. Model-driven techniques for evaluating the qos of middleware configurations for dre systems. In Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium, pages 180–189, San Francisco, CA, March 2005.

[38] Arvind S. Krishna, Nanbor Wang, Balachandran Natarajan, Anniruddha Gokhale, Douglas C. Schmidt, and Gautam 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.

[39] Kun-Chan Lan and John Heidemann. Rapid model parameterization from traffic measurements. ACM Transactions on Modeling and Computer Simulation, 12(3):201–229, July 2002.

[40] Ben Laurie and Peter Laurie. Apache: The Definitive Guide. O'Reilly and Associates, 3 edition, 2002.

[41] Lawrence M. Leemis and Stephen K. Park. Discrete-Event Simulation: A First Course. Prentice Hall, 2005.

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

[43] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. Parallel Computing, 30(7):817–840, 2004.

[44] 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.

[45] Toshiyuki Miyachi, Ken ichi Chinen, and Yoichi 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 (Tridentcom '05), pages 274–282, Trento, Italy, February 2005.

[46] John D. Musa. Operational profiles in software-reliability engineering. IEEE Software, 10(2):14–32, March 1993.

[47] Erich M. Nahum, Marcel-Catalin Rosu, Srinvasan Seshan, and Jussra Almeida. The effects of wide-area conditions on WWW server performance. In Sigmetrics, 2001.

[48] Open Software Associates. OpenWEB netDeploy, 1998.

[49] David Oppenheimer, Vitaliy Vatkovskiy, and David 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 (FuDiCo II): Survivability: Obstacles and Solutions, Bertinoro, Italy, June 2004.

[50] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the Internet. ACM SIGCOMM Computer Communication Review, 33(1):59–64, 2003.

[51] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Transactions on Computer Systems, 21(2):164–206, May 2003.

[52] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. ACM Computer Communication Review, 27(1):31–41, January 1997.

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

[54] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '01), pages 149–160, San Diego, California, August 2001.

[55] Gerald Craig Sudipto Ghosh, Nishant Bawa and Ketaki 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, October 2001.

[56] Linda Tauscher and Saul 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.

[57] Joe Touch and Steve Holtz. The X-Bone. In Third Global Internet Mini-Conference at Globecom 1998, pages 59–68, November 1998.

[58] Philip Tree. Network simulation of ip and atm over ip using a discrete event simulator. Master's thesis, University of Waikato, Hamilton, New Zealand, July 1999.

[59] Péter Urbán, Xavier Défago, and André Schiper. Neko: A single environment to simulate and prototype distributed algorithms. Journal of Information Science and Engineering, 17(6):981–997, November 2002.

[60] Yu-Shun Wang and Joe Touch. Application deployment in virtual networks using the X-Bone. In DARPA: Active Networks Conference and Exposition (DANCE 2002), pages 484–493, 2002.

[61] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet 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, December 2002. USENIX Association.

[62] Alec Wolman, Geoffrey M. Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry 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, December 1999.

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

[64] Weiwei Song W.T. Tsai, Ray Paul and Zhibin Cao. Coyote: An xml-based framework for web serivces testing. In Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering (HASE '02), Tokyo, Japan, October 2002.

[65] Cemal Yilmaz, Atif Memon, Adam Porter, Arvind S. Krishna, Douglas C. Schmidt, Aniruddha Gokhale, , and Balachandran Natarajan. Preserving distributed system's critical properties–a model-driven approach. IEEE Software Special Issue on the Persistent Software Attributes, 21(6):32–40, 2004.

[66] M. Yuksel, B. Sikdar, K. S. Vastola, and B. Szymanski. Workload generation for ns simulations of wide area networks and the internet. In the Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS) part of Western Multi-Conference (WMC), San Diego, CA, 2000.

[67] Cliff Changchun Zou, Weibo Gong, and Don Towsley. Code red worm propagation modeling and analysis. In CCS02. Association for Computer Machinery, November 2002.

# Appendix A

## Declaration Macros for Weevil's Workload Configuration Model

The following macros are used to declare and configure the workload scenario conceptual model. For each declaration macro, the following information is listed:

- declaration macro name
- arguments of the macro
- the property macros defined by the declaration macro

```
*WVL_SYS_ExternalLibrary -- [optional] Represents a library dependency that the workload
 simulation program has. This allows for other code to be linked in.
 Arguments
 ---------
 $1 ID : [required] string
 $2 cflags : [optional] string
 $3 libs : [optional] string
 $4 path : [optional] string
 Definitions
 -----------
 WVL_Actor_<ID>
 WVL_Actor_<ID>_cflags = <cflags>
 WVL_Actor_<ID>_libs = <libs>
 WVL_Actor_<ID>_path = <path>

*WVL_SYS_WorkloadSimulation -- [required] Represents the whole workload simulation.
 Arguments
 ---------
 $1 ID : [required] string
 $2 length : [optional] int
 $3 processes : [required] list<SimulationProcess.ID>
 Definitions
 -----------
 WVL_WorkloadSimulation_<ID>
 WVL_WorkloadSimulation_<ID>_length = <length>
 WVL_WorkloadSimulation_<ID>_processes = <processes>

*WVL_SYS_SimulationProcess -- [required] Represents a single simulation process.
```

```
 Arguments
 ---------
 $1 ID : [required] string
 $2 type : [required] SimulationProcessType
 Definitions
 -----------
 WVL_SimulationProcess_<ID>
 WVL_SimulationProcess_<ID>_type = <type>

*WVL_SYS_SimulationProcessProp -- [optional] Represents a property associated with a
 simulation process.
 Arguments
 ---------
 $1 simulationProcessID : [required] SimulationProcess.ID
 $2 propertyName : [required] string
 $3 propertyValue : [required] string
 Definitions
 -----------
 WVL_SimulationProcess_<simulationProcessID>_<propertyName> = <propertyValue>
 *WVL_SYS_Insert( ‘WVL_SimulationProcess_<simulationProcessID>_props’, ‘<propertyName>’)

*WVL_SYS_SimulationProcessType -- [required] Represents a class of simulation processes.
 Arguments
 ---------
 $1 ID : [required] string
 $2 sourceFiles : [required] list<string>
 $3 libraries : [optional] list<ExternalLibrary.ID>
 Definitions
 -----------
 WVL_SimulationProcessType_<ID>
 WVL_SimulationProcessType_<ID>_sourceFiles = <sourceFiles>
 WVL_SimulationProcessType_<ID>_libraries = <libraries>
```

# Appendix B

## Declaration Macros for Weevil's Trial Configuration Model

The following macros are used to declare and configure the trial conceptual models. For each declaration macro, the following information is listed:

- declaration macro name
- arguments of the macro
- the property macros defined by the declaration macro (these property macros can be used in other declaration macro's arguments)

```
*WVL_SYS_Actor -- [required] Represents an instance of an ActorType (user actor or
 environment actor) that supplies the requests associated with a particular simulation
 process to a particular trial entity.
 Arguments
 ---------
 $1 ID : [required] string
 $2 simulationProcess : [required] string
 $3 program : [required] ActorType.ID
 $4 entity : [required] enum{Component.ID, Host.ID}
 $5 dir : [optional] string
 Definitions
 -----------
 WVL_Actor_<ID>
 WVL_Actor_<ID>_simulationProcess = <simulationProcess>
 WVL_Actor_<ID>_program = <program>
 WVL_Actor_<ID>_entity = <entity>
 WVL_Actor_<ID>_dir = <dir>

*WVL_SYS_ActorType -- [required] Represents a type of actors that share the same formats of
 interpretation programs to interpret and pass the actions through a component running
 locally to it and message receiving programs to interpret the inbound messages.
 Arguments
 ---------
 $1 ID : [required] string
 $2 style : [required] enum{``Java'', ``Shell''}
 $3 binaryDistDir : [required] string
 $4 interpretProgram : [required] string
 $5 argument : [optional] string
 $6 classpath : [optional], only for Java style string
 $7 receiveProgram : [optional] string
 $8 processing : [optional] string
 $9 logs : [optional] string
 Definitions
 -----------
 WVL_ActorType_<ID>
 WVL_ActorType_<ID>_style = <style>
 WVL_ActorType_<ID>_binaryDistDir = <binaryDistDir>
 WVL_ActorType_<ID>_interpretProgram = <interpretProgram>
 WVL_ActorType_<ID>_argument = <argument>
 WVL_ActorType_<ID>_classpath = <classpath>
 WVL_ActorType_<ID>_receiveProgram = <receiveProgram>
 WVL_ActorType_<ID>_processing = <processing>
 WVL_ActorType_<ID>_logs = <logs>

*WVL_SYS_ActorProp -- [optional] Represents a property of an actor.
 Arguments
 ---------
 $1 actorID : [required] Actor.ID
 $2 propertyName : [required] string
 $3 propertyValue : [required] string
 Definitions
 -----------
 WVL_Actor_<actorID>_<propertyName> = <propertyValue>
 WVL_SYS_Insert( 'WVL_Actor_<actorID>_props', '<propertyName>')

*WVL_SYS_Component -- [required] Represents a component of the System Under Experiment (SUE).
 Arguments
 ---------
 $1 ID : [required] string
 $2 type : [required] ComponentType.ID
 $3 prestart : [required] bool
 Definitions
 -----------
 WVL_Component_<ID>
 WVL_Component_<ID>_type = <type>
 WVL_Component_<ID>_prestart = <prestart>

*WVL_SYS_ComponentHost -- [required] Binds an SUE component to a testbed host.
 Arguments
 ---------
 $1 ID : [required] string
 $2 component : [required] Component.ID
 $3 host : [required] Host.ID
 Definitions
 -----------
 WVL_ComponentHost_<ID>
 WVL_ComponentHost_<ID>_component = <component>
 WVL_ComponentHost_<ID>_host = <host>
 WVL_ComponentHost_<component>_host = <host>

*WVL_SYS_ComponentHostProp -- [optional] Property associated with a ComponentHost combination.
 Arguments
 ---------
 $1 componentHostID : [required] ComponentHost.ID
 $2 propertyName : [required] string
 $3 propertyValue : [required] string
 Definitions
 -----------
 WVL_ComponentHost_<componentHostID>_<propertyName> = <propertyValue>
 WVL_SYS_Insert( 'WVL_ComponentHost_<componentHostID>_props', '<propertyName>')

*WVL_SYS_ComponentHostType -- [optional] Binds a ComponentType to a HostType.
```

```
 Arguments
 ---------
 $1 componentTypeID : [required] ComponentType.ID
 $2 hostTypeID : [required] HostType.ID
 $3 binaryDistDir : [required] string
 Definitions
 -----------
 WVL_ComponentHostType_<componentTypeID>_<hostTypeID>
 WVL_ComponentHostType_<componentTypeID>_<hostTypeID>_binaryDistDir = <binaryDistDir>

*WVL_SYS_ComponentHostTypeProp -- [optional] Represents a property associated with a
 ComponentHostType combination.
 Arguments
 ---------
 $1 componentTypeID : [required] ComponentType.ID
 $2 hostTypeID : [required] HostType.ID
 $3 propertyName : [required] string
 $4 propertyValue : [required] string
 Definitions
 -----------
 WVL_ComponentHostType_<componentTypeID>__<hostTypeID>_<propertyName> = <propertyValue>
 WVL_SYS_Insert( ‘WVL_ComponentHostType_<componentHostTypeID>_<hostTypeID>_props’,
  ‘<propertyName>’)

*WVL_SYS_ComponentOrder -- [optional] Represents the sequence order to start SUE components.
 Arguments
 ---------
 $1 ID : [required] string
 $2 sequence : [required] string
 Definitions
 -----------
 WVL_ComponentOrder_<ID>
 WVL_ComponentOrder_<ID>_sequence = <sequence>

*WVL_SYS_ComponentProp -- [optional] Represents a property of an SUE component.
 Arguments
 ---------
 $1 componentID : [required] Component.ID
 $2 propertyName : [required] string
 $3 propertyValue : [required] string
 Definitions
 -----------
 WVL_Component_<componentID>_<propertyName> = <propertyValue>
 WVL_SYS_Insert( ‘WVL_Component_<componentID>_props’, ‘<propertyName>’)

*WVL_SYS_ComponentRelation -- [optional] Named binary relation between two SUE components.
 Arguments
 ---------
 $1 ID : [required] string
 $2 name : [required] string
 $3 src : [required] Component.ID
 $4 dest : [required] Component.ID
 Definitions
 -----------
 WVL_ComponentRelation_<ID>
 WVL_ComponentRelation_<ID>_name = <name>
 WVL_ComponentRelation_<ID>_src = <src>
 WVL_ComponentRelation_<ID>_dest = <dest>
 WVL_ComponentRelation_<src>_<ID> = <dest>

*WVL_SYS_ComponentRelationProp -- [optional] Represents a property associated with a component
relation.
 Arguments
 ---------
 $1 componentRelationID : [required] ComponentRelation.ID
 $2 propertyName : [required] string
 $3 propertyValue : [required] string
 Definitions
 -----------
 WVL_ComponentRelation_<componentRelationID>_<propertyName> = <propertyValue>
 WVL_SYS_Insert( ‘WVL_ComponentRelation_<componentRelationID>_props’, ‘<propertyName>’)

*WVL_SYS_ComponentType -- [required] Represents a type of components that share the same
 APIs.
 Arguments
 ---------
 $1 ID : [required] string
 $2 startScript : [required] string
 $3 stopScript : [required] string
 $4 startArgs : [optional] string
 $5 config : [optional] string
 $6 processing : [optional] string
 $7 logs : [optional] string
 Definitions
 -----------
 WVL_ComponentType_<ID>
 WVL_ComponentType_<ID>_startScript = <startScript>
 WVL_ComponentType_<ID>_stopScript = <stopScript>
 WVL_ComponentType_<ID>_startArgs = <startArgs>
 WVL_ComponentType_<ID>_config = <config>
 WVL_ComponentType_<ID>_processing = <processing>
 WVL_ComponentType_<ID>_logs = <logs>

*WVL_SYS_ComponentTypeProp -- [optional] Represents a property of a component type.
 Arguments
 ---------
 $1 componentTypeID : [required] ComponentType.ID
 $2 propertyName : [required] string
 $3 propertyValue : [required] string
 Definitions
 -----------
 WVL_ComponentType_<componentTypeID>_<propertyName> = <propertyValue>

*WVL_SYS_Host -- [optional] Represents a single host of the testbed.
 Arguments
 ---------
 $1 ID : [required] string
 $2 address : [required] string
 $3 account : [required] string
 $4 weevilRoot : [required] string
 $5 bourneShell : [required] string
 $6 java : [required] string
 $7 type : [required] HostType.ID
 Definitions
 -----------
 WVL_Host_<ID>
 WVL_Host_<ID>_address = <address>
 WVL_Host_<ID>_account = <account>
 WVL_Host_<ID>_weevilRoot = <weevilRoot>
 WVL_Host_<ID>_bourneShell = <bourneShell>
 WVL_Host_<ID>_java = <java>
 WVL_Host_<ID>_type = <type>

*WVL_SYS_HostProp -- [optional] Represents a property of a testbed host.
 Arguments
 ---------
 $1 hostID : [required] Host.ID
 $2 propertyName : [required] string
 $3 propertyValue : [required] string
 Definitions
 -----------
 WVL_Host_<hostID>_<propertyName> = <propertyValue>
 WVL_SYS_Insert( ‘WVL_Host_<hostID>_props’, ‘<propertyName>’)
```

```
*WVL_SYS_HostType -- [required] Represents a class of hosts.
 Arguments
 ---------
 $1 ID : [required] string
 Definitions
 -----------
 WVL_HostType_<ID>

*WVL_SYS_SUE -- [required] Represents the System Under Experiment (SUE).
 Arguments
 ---------
 $1 ID : [required] string
 $2 components : [required] list<Component.ID>
 $3 relations : [optional] list<ComponentRelation.ID>
 $4 order : [optional] ComponentOrder.ID
 Definitions
 -----------
 WVL_SUE_<ID>
 WVL_SUE_<ID>_components = <components>
 WVL_SUE_<ID>_relations = <relations>
 WVL_SUE_<ID>_order = <order>

*WVL_SYS_Testbed -- [required] Represents the collection of hosts that comprises the testbed.
 Arguments
 ---------
 $1 ID : [required] string
 $2 hosts : [required] list<Host.ID>
 Definitions
 -----------
 WVL_Testbed_<ID>
 WVL_Testbed_<ID>_hosts = <hosts>
 WVL_SYS_Insert( ‘WVL_ComponentType_<componentTypeID>_props’, ‘<propertyName>’)

*WVL_SYS_Trial -- [optional] Represents a trial, highest-level macro in all the macros
 Arguments
 ---------
 $1 ID : [required] string
 $2 userworkload : [required] Workload.ID
 $3 useractors : [required] list<Actor.ID>
 $4 environmentworkload : [optional] Workload.ID
 $5 environmentactors : [optional] list<Actor.ID>
 $6 testbed : [required] Testbed.ID
 $7 sue : [required] SUE.ID
 $8 componentHosts : [required] list<ComponentHost.ID>
 $9 parallel : [required] integer
 $10 timeout : [required] integer
 $11 clean : [optional] bool
 Definitions
 -----------
 WVL_Trial_ID = <ID>
 WVL_Trial_<ID>
 WVL_Trial_<ID>_userworkload = <userworkload>
 WVL_Trial_<ID>_useractors = <useractors>
 WVL_Trial_<ID>_environmentworkload = <environmentworkload>
 WVL_Trial_<ID>_environmentactors = <environmentactors>
 WVL_Trial_<ID>_testbed = <testbed>
 WVL_Trial_<ID>_sue = <sue>
 WVL_Trial_<ID>_componentHosts = <componentHosts>
 WVL_Trial_<ID>_parallel = <parallel>
 WVL_Trial_<ID>_timeout = <timeout>
 WVL_Trial_<ID>_clean = <clean>

*WVL_SYS_Workload -- [required] Represents the user workload or the environment workload.
 Arguments
 ---------
 $1 ID : [required] string
 $2 filename : [required] string
```

```
$3 processes : [required] list<string>
Definitions
-----------
WVL_SUE_<ID>
WVL_SUE_<ID>_filename = <filename>
WVL_SUE_<ID>_processes = <processes>
```

# Appendix C

## Weevil-Defined Property Macros

The following property macros are defined in Weevil by a combination of more than one declaration macro functions. These property macros, together with those defined by each declaration macro function as shown in Appendix B, could be used to configure the **ComponentType** or the **ActorType**. For each property macro, the following information is listed:

- meaning of the macro
- the definition of the macro

The **ComponentType** configuration is mainly used to provide generally applicable templates for each component of this type. The macro **WVL_Component_ID** represents any component of this type. Besides, the following macros have been defined in Weevil to be used in **ComponentType** configuration.

```
*WVL_Trial_Name
 Meaning
 -------
 the name of the current experiment trial
 Definition
 ----------
 globally defined

*theSUE
 Meaning
 -------
 the SUE in the current trial
 Definition
 ----------
 ‘WVL_Trial_’WVL_Trial_ID‘_sue’

*WVL_Component_ID
 Meaning
 -------
 the identity of a specific component
 Definition
 ----------
 globally defined

*theHost
 Meaning
 -------
 the host ID that the current component is located at
 Definition
 ----------
 ‘WVL_ComponentHost_’WVL_Component_ID‘_host’

*theAddress
 Meaning
 -------
 the host address of the component
 Definition
 ----------
 ‘WVL_Host_’theHost‘_address’

*theAccount
 Meaning
 -------
 the host account of the component
 Definition
 ----------
 ‘WVL_Host_’theHost‘_account’

*theWeevilRoot
 Meaning
 -------
 the workspace on the host of the component
 Definition
 ----------
 ‘WVL_Host_’theHost‘_weevilRoot’

*theType
 Meaning
 -------
 the type of the current component
 Definition
 ----------
 ‘WVL_Component_’WVL_Component_ID‘_type’

*theCompPath
 Meaning
 -------
 the directory path on the host for the current component where all the component-related
  scripts and data are deployed to
 Definition
 ----------
 theWeevilRoot/WVL_Trial_Name/theType/WVL_Component_ID

*theSoftwarePath
 Meaning
 -------
 the directory path on the host for the current component where the component type’s binary
  distributions are deployed to
 Definition
 ----------
 theWeevilRoot/WVL_Trial_Name/theType/software
```

The **ActorType** configuration is mainly used to provide generally applicable templates for each actor of this type. The macro **WVL_Actor_ID** represents any actor of this type. Besides, the following macros have been defined in Weevil to be used in **ActorType** configuration.

```
*WVL_Trial_Name
 Meaning
 -------
 the name of the current experiment trial
 Definition
 ----------
 globally defined

*theSUE
 Meaning
 -------
 the SUE in the current trail
 Definition
 ----------
 `WVL_Trial_'WVL_Trial_ID`_sue'

*WVL_Actor_ID
 Meaning
 -------
 the identity of a specific actor
 Definition
 ----------
 globally defined

*theEntity
 Meaning
 -------
 the identity of the entity that the current actor is associated to
 Definition
 ----------
 `WVL_Actor_'WVL_Actor_ID`_entity'

*theHost
 Meaning
 -------
 the host ID that the current actor is located at
 Definition
 ----------
 ifelse(`WVL_SYS_Contains(`theEntity', theComponents)', `Y', `WVL_ComponentHost_'theEntity`_host',
  theEntity)

*theAddress
 Meaning
 -------
 the host address of the actor
 Definition
 ----------
 `WVL_Host_'theHost`_address'
```

```
*theAccount
 Meaning
 -------
 the host account of the actor
 Definition
 ----------
 `WVL_Host_'theHost`_account'

*theWeevilRoot
 Meaning
 -------
 the workspace on the host of the actor
 Definition
 ----------
 `WVL_Host_'theHost`_weevilRoot'

*theTimeStampScript
 Meaning
 -------
 the shell script Weevil provides to get the trial-execution-time stamp
 Definition
 ----------
 theWeevilRoot/WVL_Trial_Name`_'theHost`_timestamp.sh

*theType
 Meaning
 -------
 the type of the current actor
 Definition
 ----------
 `WVL_Actor_'WVL_Actor_ID`_type'

*theEntityType
 Meaning
 -------
 the type of the entity that the actor is associated to
 Definition
 ----------
 ifelse(`WVL_SYS_Contains(`theEntity', theComponents)', `Y', `WVL_Component_'theEntity`_type',
  `WVL_Host_'theEntity`_type')

*theActorPath
 Meaning
 -------
 the directory path on the host for the current actor where all the actor-related scripts and data
  are deployed to
 Definition
 ----------
 theWeevilRoot/WVL_Trial_Name/theType/WVL_Actor_ID

*theEntityPath
 Meaning
 -------
 the directory path on the host for the actor-associated entity
 Definition
 ----------
 ifelse(`WVL_SYS_Contains(`theEntity', theComponents)', `Y',
  `theWeevilRoot/WVL_Trial_Name/theType/WVL_Component_ID', `theWeevil/WVL_Trial_Name')

*theSoftwarePath
 Meaning
 -------
 if the associated entity is an SUE component, then it represents the directory path on the host where
  the associated component type's binary distributions are deployed to
 Definition
 ----------
 theWeevilRoot/WVL_Trial_Name/theEntityType/software
```

# Appendix D

## Weevil-Defined Functional Macros

The following functional macros are defined in Weevil for the software engineers to ease their trial configuration. For each functional macro, the following information is listed:

- functionality of the macro
- an example and the expanded result using the macro

```
*WVL_SYS_Range(string, int, int)
 Function
 --------
 A name list, the resulting list is $1$2, $1($2+1), ...$1$3
 Example
 -------
 WVL_SYS_Range(‘br’, 1, 5)
 => br1,br2,br3,br4,br5

*WVL_SYS_Echo(name)
 Function
 --------
 Display the value of the argument
 Sometimes, when a marco expansion is within another macro expansion,
  it won’t be expanded automatically. To solve the problem, we define
  this macro to ask the inner macro to expand first.
 Example
 -------
 Assuming that:
 WVL_Component_ID = S0
 WVL_SYS_Host(‘H0’, ‘skylark.cs.colorado.edu’, ‘ywang’, ‘’, ‘/bin/sh’,
  ‘/scratch/yanyan/testroot’, ‘FreeBSD’)
 WVL_SYS_Component(‘S0’, ‘SienaServer’)
 WVL_SYS_Actor(‘D1’, ‘D1’, ‘DrProgram1’, ‘S0’)
 WVL_SYS_Echo(‘WVL_Component_’WVL_SYS_Echo(‘WVL_Component_’WVL_Component_ID‘_Parent’)‘_type’)
 => skylark.cs.colorado.edu

*WVL_SYS_Head(string[, string]*)
 Function
 --------
 Returns the head of its argument list, i.e., the first string
 Example
 -------
 WVL_SYS_Head(‘S0’, ‘S1’, ‘S2’)
 => S0

*WVL_SYS_Tail(string[, string]*)
 Function
 --------
 Returns the argument list except the head
 Example
 -------
 WVL_SYS_Tail(‘S0’, ‘S1’, ‘S2’)
 => S1,S2

*WVL_SYS_Add(list, string[, string])
 Function
 --------
 Defines or redefines $1 to be its previous contents with $2 appended
 Example
 -------
 define(‘Group1’, ‘S1,S2’)
 define(‘Group2’, ‘S3,S4’)
 WVL_SYS_Add(‘Group2’, Group1)
 Group2
 => S3,S4,S1,S2

*WVL_SYS_Contains(string, list[, list]*)
 Function
 --------
 Checks if $1 is contained by the list represented by the rest of the arguments
 Example
 -------
 define(‘Group1’, ‘S1,S2’)
 define(‘Group2’, ‘S3,S4’)
 WVL_SYS_Contains(‘S3’, Group1)
 => false
 WVL_SYS_Contains(‘S3’, Group1, Group2)
 => true

*WVL_SYS_Insert(list, string)
 Function
 --------
 Adds $2 to $1 if it doesn’t already exist there
 Example
 -------
 define(‘Group1’, ‘S1,S2’)
 WVL_SYS_Insert(‘Group1’, ‘S1’)
 Group1
 => S1,S2
 WVL_SYS_Insert(‘Group1’, ‘S3’)
 Group1
 => S1,S2,S3

*WVL_SYS_Callforeach(function, string [, string])
 Function
 --------
 Applies $1 to $2 and then processes the rest one by one, $1 can only accept one argument
 Example
 -------
 define(‘serverdef’, ‘WVL_SYS_Component($1, ‘SienaServer’)’)
 WVL_SYS_Callforeach(‘serverdef’, S0, S1, S2)
 => WVL_SYS_Component(‘S0’, ‘SienaServer’)dnl
 WVL_SYS_Component(‘S1’, ‘SienaServer’)dnl
 WVL_SYS_Component(‘S2’, ‘SienaServer’)dnl

*WVL_SYS_Foreach(name, function, list)
```

```
 Function
 --------
 Iterates over $3 defining $1 to the current value in $3 and executing $2
 Example
 -------
 WVL_SYS_Foreach(`i', `WVL_SYS_Component(i, `SienaServer')', `S0, S1, S2')
 => WVL_SYS_Component(`S0', `SienaServer')dnl
 WVL_SYS_Component(`S1', `SienaServer')dnl
 WVL_SYS_Component(`S2', `SienaServer')dnl

*WVL_SYS_Fortwo(name, name, function, string[, string]*)
 Function
 --------
 Iterates over $4 defining $1 to the current value and $2 to the next value and executing $3
 Example
 -------
 WVL_SYS_Fortwo( `i', `j', ``i' is set to i, `j' is set to j', `a', `b', `123', `this,is')
 => i is set to a, j is set to b
 i is set to 123, j is set to this,is

*WVL_SYS_Fortwovar(name, name, function, list, list)
 Function
 --------
 Iterates over $4 and $5 defining $1 and $2 to the current value of $4 and $5 and executing $3
 Example
 -------
 WVL_SYS_Fortwo( `i', `j', ``i' is set to i, `j' is set to j', `a, b, c', `1, 2, 3')
 => i is set to a, j is set to 1
 i is set to b, j is set to 2
 i is set to c, j is set to 3

*WVL_SYS_First(list)
 Function
 --------
 return the first item in $1
 Example
 -------
 WVL_SYS_First( `1, 2, 3')
 => 1

*WVL_SYS_Subst_Commas(string, string [, string])
 Function
 --------
 Iterates over arguments replacing commas with $1
 Example
 -------
 WVL_SYS_Subst_Commas(`;', S0, S1, S2)
 => S0;S1;S2;

*WVL_SYS_Strip_Commas(string [, string]*)
 Function
 --------
 Iterates over arguments expanding without commas, equals to
 WVL_SYS_Subst_Commas(` ', name [, name])
 Example
 -------
 WVL_SYS_Strip_Commas(S0, S1, S2)
 => S0 S1 S2
```