

*The Defense Advanced Research Projects Agency Information Technology Office funded the work in this thesis under the contracts F30602-96-C-0315, F30602-97-C-0276, and F30602-98-C-0187.*



© Copyright by Ramesh U.V. Chandra, 2001

LOKI: A STATE-DRIVEN FAULT INJECTOR FOR DISTRIBUTED  
SYSTEMS

BY

RAMESH U.V. CHANDRA

B. Tech., Indian Institute of Technology, Madras, 1998

THESIS

Submitted in partial fulfillment of the requirements  
for the degree of Master of Science in Computer Science  
in the Graduate College of the  
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

मातृ देवोभवः पितृ देवोभवः आचार्य देवोभवः

*To my mother, father, and teachers, who are equivalent to God on earth...*

# Acknowledgements

I would like to thank my advisor Prof. William H. Sanders for his invaluable guidance and support and for always being there to discuss and clarify any matter, be it on the academic front or the research front. I would also like to thank Dr. Michel Cukier for the excellent role he played as an additional research advisor by acting as a good sounding board for my ideas.

I owe a lot to the research atmosphere at the PERFORM lab, and I thank all the members of the lab, namely, Ryan Lefever, Kaustubh Joshi, Jennifer Ren, Paul Rubel, Sudha Krishnamurthy, Robert Waldrop, Jay Doyle, David Daly, Dan Deavours, Patrick Webster, and Amy Christensen, for having created a very conducive atmosphere for me to work in. In particular, I would like to thank my research partner, Ryan Lefever, who patiently put up with all the long discussions on research issues. Working with him as a team was a pleasure and he played a key role in making Loki a complete fault injection tool. I would like to thank all my friends in Champaign for providing wonderful company during my stay there. I thank Jenny Applequist for her invaluable help in administrative matters, and for the help in “which-hunting,” among other advice, during the preparation of my technical documents.

Last but certainly not the least, I would like to thank Amma, Daddy, Anna, Bujji, and Prahladh.

# Abstract

Distributed systems are being increasingly used to build critical systems. This necessitates the validation of their dependability. Fault injection, using a representative set of faults, is an important and effective method of validating dependable systems. However, a distributed system can fail in subtle ways that depend on the state of multiple parts of the system. This suggests that faults should be injected in a distributed system based on its the global state. However, it is well known that it is practically impossible to maintain the global state of a distributed system at runtime, with minimal intrusion into the system. Hence, fault injection based on the global state of a distributed system is difficult.

This thesis presents work on a fault injector, called Loki, that addresses the challenges of global-state-based fault injection in distributed systems. In Loki, fault injection is performed based on a partial view of the global state of a distributed system. Once faults are injected, a post-runtime analysis, using off-line clock synchronization, is used to place events and injections on a single global timeline and to determine whether the intended faults were properly injected. Finally, experiments containing successful fault injections are used to estimate the specified measures. The contributions of the work in this thesis include an enhanced Loki runtime that allows dynamic entry and exit of nodes in the system, and a new and flexible method for obtaining a wide range of performance and dependability measures in Loki.

# Table of Contents

<b>Chapter 1 Introduction</b>	<b>1</b>
1.1 Motivation	1
1.2 Related Work	3
1.3 Organization of the Thesis	6
<b>Chapter 2 Overview of the Loki Fault Injector</b>	<b>7</b>
2.1 Introduction	7
2.2 Loki Concepts	8
2.2.1 Partial View of Global State	8
2.2.2 Node	9
2.2.3 Fault Injection Campaign	9
2.3 Evaluation of a System Using Loki	10
2.4 The Runtime Phase	10
2.5 Analysis Phase	13
2.6 Measure Estimation Phase	16
<b>Chapter 3 Loki Runtime Architecture</b>	<b>18</b>
3.1 Introduction	18
3.2 The Previous Loki Runtime	18
3.2.1 Overview of the Previous Runtime	18
3.2.2 Performance Analysis	19
3.3 Shortcomings of the Original Runtime	21
3.4 Design Choices for the Enhanced Loki Runtime	21
3.4.1 The Design Choices	22
3.4.2 Comparison of the Design Choices	23
3.5 Architecture of the New Loki Runtime	26
3.5.1 Central Daemon	26
3.5.2 Local Daemon	28
3.5.3 State Machine	30
3.5.4 State Machine Transport	32
3.5.5 Fault Parser	32
3.5.6 Recorder	34
3.5.7 Probe	36
3.6 Operation of the New Loki Runtime	37

3.6.1	Normal Operation . . . . .	37
3.6.2	On a Node Crash . . . . .	39
3.6.3	On a Node Restart . . . . .	39
3.6.4	On a Host Crash and Reboot . . . . .	40
3.7	Benefits of the Enhanced Runtime Architecture . . . . .	40
3.8	Limitations of the Current Loki Implementation . . . . .	41
<b>Chapter 4</b>	<b>Measure Estimation in Loki . . . . .</b>	<b>42</b>
4.1	Introduction . . . . .	42
4.2	Overview of Measure Estimation in Loki . . . . .	42
4.3	Measures Defined at the Study Level . . . . .	45
4.3.1	Predicate . . . . .	45
4.3.2	Observation Function . . . . .	47
4.3.3	Subset Selection . . . . .	48
4.3.4	Study-Level Measures . . . . .	49
4.4	Measures Defined Across Studies . . . . .	50
4.4.1	Simple Sampling Measures . . . . .	50
4.4.2	Stratified Weighted Measures . . . . .	51
4.4.3	Stratified User Measures . . . . .	53
<b>Chapter 5</b>	<b>Example of a Fault Injection Campaign . . . . .</b>	<b>54</b>
5.1	Introduction . . . . .	54
5.2	Test Application . . . . .	54
5.3	State Machine Specification . . . . .	55
5.4	Fault Specification . . . . .	60
5.5	Instrumentation and Probe Design . . . . .	61
5.6	Campaign Execution . . . . .	63
5.7	Campaign Analysis . . . . .	65
5.8	Measure Specification and Estimation . . . . .	66
<b>Chapter 6</b>	<b>Conclusions . . . . .</b>	<b>69</b>
<b>References</b>	<b>. . . . .</b>	<b>71</b>

# List of Figures

2.1	Evaluation of a System using Loki . . . . .	11
3.1	Original Loki Runtime Architecture . . . . .	19
3.2	Correct Fault Injection Probability as a Function of Time Spent in a State (10ms Linux Timeslice) . . . . .	20
3.3	Correct Fault Injection Probability as a Function of Time Spent in a State (1ms Linux Timeslice) . . . . .	21
3.4	Design Choices for the Loki Runtime Architecture . . . . .	23
3.5	The New Loki Runtime Architecture . . . . .	27
4.1	Measure Estimation in Loki . . . . .	43
4.2	Predicate Value Timeline Example . . . . .	46
5.1	Election Protocol . . . . .	56

# Chapter 1

## Introduction

### 1.1 Motivation

The dependability of computer systems is very important, considering the pervasiveness of computers in our everyday life. However, the level of dependability required varies from system to system; for example, a computer system controlling a medical system must be much more dependable than a web server, which should be more dependable than a home user's desktop. Thus, each computer system should be validated to ensure that it meets the required dependability levels.

Before moving further, a few definitions are in order. A *fault* is any physical influence that could potentially cause the system to deviate from its intended functionality. An *error* is the effect of the fault on the system and thus is the result of the fault. If the system is not able to cope with the error then a *failure* of the system results [1]. Note that not every fault leads to an error and not every error results in a failure. *Coverage* is the probability that the system recovers (i.e., does not have a failure) given that a fault occurred. *Dormancy* is the amount of time between the occurrence of a fault and its manifestation as an error. *Latency* is the amount of time from the occurrence of an error to either failure of the system or detection and system recovery. Coverage, dormancy, and latency are important properties of a dependable system that need to be evaluated.

Ideally, validating a system should involve running it in a configuration and environment

similar to that in the field for sufficiently long enough to determine the system's dependability. However, this is not possible in practice. Therefore, in practice, two methods are used: system modeling and fault injection.

System modeling involves developing a formal mathematical representation of the system and solving it to determine the system's properties. It requires a detailed knowledge of the complete system and its execution. Modeling has the advantage that the behavior of the system over a long period of time, and in a variety of configurations and environments, can be obtained in a relatively short period of time. However, by its very nature, a model cannot mimic the actual system completely; hence, results from modeling could be less accurate than those from runs of the actual system. Also, some parameters needed by a system model have to be obtained from the system itself. In practice, these parameters are very sensitive, and an inaccurate estimation would lead to wrong overall values for the properties of the dependable system.

Experimental evaluation of the system and, in particular, fault injection can be used to obtain the required parameters. Fault injection is the process of generating faults in a system at a rate much higher than the rate at which they appear in the system's actual environment. This is done because it would take too long to wait for faults to occur by themselves in the system. From the generated faults, observations are collected that are used to determine the system's properties. A tool that helps in the process of fault injection is called a *fault injector*. Fault injection is used for two different purposes: *fault removal* and *fault forecasting*.

Fault removal uses fault injection to stress the system under study to verify whether the system behaves as given in its functional specification. Thus it is used during the building of the system to remove any design and implementation deficiencies ("bugs") in the system. Fault forecasting is done after the system has been completely implemented. It uses fault injection to evaluate the efficiency of the long-term operational behavior of the fault-tolerance mechanisms in the system. One good way of using modeling and fault injection together

to evaluate a system is as follows. First, a model of the system design is developed. Then a prototype is implemented and evaluated to obtain the parameters needed for the model. The parameters are then fit into the model to check whether the system design is adequate. If not, the design is modified and the above process is repeated until the design is adequate. The final system design is then implemented and fault-injected for fault removal. Finally, fault forecasting is done on the system before it is deployed.

If the system under consideration is a reliable distributed system, a failure of a component in the system could result from a fault in the component itself or from the propagation of a fault in another component of the system. Thus, failures in the system depend on the global execution state of the system, i.e., the combined state of all the components in the system. For that reason, during fault injection of a reliable distributed system, a fault injector needs to keep track of the global state of the system. Other desirable features of fault injection in a distributed system include verification that the faults have been injected in the intended global states, and a flexible measure estimation method.

This thesis focuses on the development of a fault injection tool, called *Loki*, to assist in the evaluation of a system's dependability, particularly when the system is a distributed system. More specifically, Loki provides a framework for specifying and injecting faults in a distributed system based on the global state of the system. The results from the fault injection experiments are collected and a check is performed, by constructing a global timeline from the results, to verify the correctness of the global-state-based fault injections. Loki also provides a mechanism for specifying and computing a wide range of dependability and performance measures from the fault injection results.

## 1.2 Related Work

There are several fault injection and measurement tools already in existence, several of which are targeted specifically to distributed systems. These tools are well-suited for the applica-

tions for which they are intended. However, they do not meet all the requirements necessary for distributed system fault injection, namely fault injection based on global state, verification of the correctness of faults, and accurate computation of a wide range of performance and dependability measures. This section gives a brief description of these tools.

JEWEL [2] is a measurement system that performs monitoring and evaluation tasks of distributed (and local) systems based on user-defined specifications. A graphical visualization tool is used to display the results on-line (i.e., in real-time). JEWEL also provides off-line analysis of the collected results. However, it can only observe the system under study; it cannot control the system's environment by injecting faults. Also, it uses hardware clocks in the system that are synchronized to maintain a global clock.

CESIUM [3] is a testing environment based on the centralized simulation of distributed executions and failures. The distributed execution of the processes in the system under study is simulated on a single machine in a single address space, with network interaction and system clocks simulated by the CESIUM environment. The code of the system under study is not instrumented; therefore, decisions and observations cannot be made based on the internal state of a process, and fault injections cannot be targeted to specific states of operation within a node. In addition, this simulation-based approach cannot fully mirror the operation of the system in a real environment; hence, for proper evaluation of the system, it is desirable to experiment on a real system. Also, no mechanism for obtaining measures is present in CESIUM.

The DOCTOR [4] fault injector is one of the tools developed to evaluate the HARTS distributed real-time system. Memory, CPU, or communication faults can be injected probabilistically or based on past history. However, faults cannot be injected based on the global state of the distributed system. Also, the correctness of the injected faults cannot be verified. DOCTOR has an integrated synthetic workload generator that can be used during the evaluation of the system. During the evaluation, performance and dependability data is collected. However, there is no mechanism for obtaining measures. Timing is done using a

hardware solution, which requires a shared backplane bus not available to all systems.

EFA [5] is designed to be used mostly for verification of the system, i.e., checking whether the system works according to its functional specification. The EFA fault injector generates random fault cases, user-defined fault cases, and/or fault cases derived from an analysis of the source program of the fault-tolerant system. It allows the user to express fault locations and types using a special language. EFA also provides support for controlling the sequence of concurrent events in a distributed system. However, EFA's approach may be too intrusive to the system. Also, it provides neither verification of the proper injection of faults nor a mechanism to obtain measures.

The Orchestra [6] fault injector integrates into the system under study as a layer that can be inserted anywhere in a layered protocol stack. Orchestra allows for fault injection based on the local state of a node; however, it does not provide a formal means for exchanging information between nodes, and does not allow fault injection based on the global state. Additionally, it does not provide a means for generating measures.

SPI [7] provides a flexible framework for distributed system evaluation and visualization. It is based on the event-action programming model, in which "ea-machines" observe events in the system and execute actions. Though SPI was developed primarily with measurement and evaluation in mind, it can also be used for fault injection. However, it can neither check for proper fault injection nor generate measures. This system currently runs on SUN workstations and the Intel Paragon.

The NFTAPE [8] fault injection tool was developed to inject both hardware- and software-based faults. To do this, the tool is divided into system-independent and system-dependent parts. The system-independent part executes the experiments, monitors the system, and collects observations. The actual fault injection is done by lightweight fault injectors that are system-dependent. This makes NFTAPE very portable. However, the tool does not verify that the fault injections have taken place in the right global states and also does not provide a means of obtaining accurate measures.

All the tools described above have been successfully applied to many systems. However, they do not have all the capabilities needed for fault injection in distributed systems. Specifically, most of them lack the ability to inject faults based on the global state of the system. Also, they do not verify that the injected faults are correct. Additionally, most of them do not have capabilities to specify and compute accurate measures from the results of the fault injections. The goal of the work described in this thesis was to design a fault injection and evaluation tool, called Loki, which has the above capabilities necessary for evaluation of distributed systems through fault injection.

### **1.3 Organization of the Thesis**

The remaining chapters describe the design of the Loki fault injector. More specifically, Chapter 2 presents the ideas underlying Loki and gives an overview of the workings of Loki. Chapter 3 describes the runtime architecture, its mechanism, and some implementation details. Chapter 4 details the mechanisms for specifying and computing a wide range of dependability and performance measures in Loki. Chapter 5 gives an example of how to use Loki for fault injection. It describes a simple example system and explains step-by-step the process of evaluating the system using Loki. Finally, Chapter 6 presents conclusions and suggestions for future work.

# Chapter 2

## Overview of the Loki Fault Injector

### 2.1 Introduction

As mentioned in Chapter 1, a fault injector for distributed systems needs to keep track of the global state of the system. However, it is well known that tracking the global state of a distributed system at runtime, with minimal intrusion into the system's behavior, is impossible in practice. A possible solution to this problem would be to synchronize the execution of all the components of the system at every state change and determine whether faults are to be injected. However, this is very intrusive to the system under study, and would cause it to deviate greatly from its normal execution. Another solution would be to have the fault injector maintain very loose synchronization by sending notifications among the components at every state change, and inject faults in a component based on the component's view of the global state. However, the component's local view of the global state could be outdated, because the system could have changed state while the state change notifications were in transit. This would cause the fault injector to inject faults in incorrect states. Measures obtained from such incorrect fault injections would not be valid.

The distributed system fault injector Loki, which is the main contribution of this thesis, solves this problem and makes fault injection based on the global state of a distributed system practical. This chapter explains the basic concepts underlying Loki and details the method by which Loki solves the above problem. Additionally, it also briefly describes the

various steps in the evaluation of a system using Loki.

## 2.2 Loki Concepts

This section describes the basic concepts of Loki. First, the concept of partial view of global state, which is central to Loki, is explained, along with the concepts of state, local state, and global state. Then the concepts of a node and Loki runtime are introduced, and the organization of a fault injection process into campaigns, studies, and experiments is explained.

### 2.2.1 Partial View of Global State

The concept of *state* is fundamental to Loki. It is assumed that at the desired level of abstraction (for fault injection and measure estimation), the execution of a component of the distributed system under study can be specified as a state machine with state transitions triggered by local events in the component. In other words, at any time, a component is in a particular local state, and it transitions to a new state only when an event occurs in it. The new state is computed using a transition function, and is uniquely determined by the old state and the event. The global state of the system is the vector of the local states of all of its components. During the fault injection process, it may be necessary to inject faults in a component based on the state of other components of the system. To do so, it is not necessary to keep track of the complete global state of the system at all times; instead, it is sufficient to track an “interesting” portion of the global state that is necessary for the injection of the required faults in the component. This interesting portion is called the *partial view of the global state*, and its selection depends on the particular system under study and the faults to be injected.

## 2.2.2 Node

In Loki, the distributed system (under study) is divided into basic components (which are processes in the current implementation), from each of which state information is collected and into each of which faults are injected. Each of these basic components of the distributed system, together with the attached Loki runtime, is called a *node*. The Loki runtime is the Loki code that executes along with the distributed system and maintains the partial view of the global state necessary for fault injections. It also performs fault injections when the system transitions to the desired states and collects information regarding state changes, fault injections, and their occurrence times. The Loki runtime can be divided into two main parts: one that is independent of the system under study, and the other that is dependent on it. The *state machine*, *state machine transport*, *fault parser*, and *recorder* constitute the system-independent part, while the *probe* is the system-dependent part. These units of the runtime together perform all of the functions that are mentioned above.

## 2.2.3 Fault Injection Campaign

Structurally, the process of fault injection using Loki consists of one or more fault injection *campaigns*. A fault injection campaign for a particular distributed system is made up of one or more *studies*. For a study, each component of the distributed system is defined by a state machine specification<sup>1</sup> and a fault specification. The state machine specification describes the execution of the component at the desired level of abstraction. It consists of the set of states, the transitions between these states, and the events in the components that trigger these transitions. The fault specification consists of a set of faults and a Boolean expression for each fault that specifies the states in which particular state machines should be when the fault is injected into the component. Details of both these specifications are

---

<sup>1</sup>Note that a state machine specification is a description that is dependent on the system under study. However, the state machine is independent of the system, and is a component of the runtime that tracks the state given in the specification.

given in Chapter 3. Note that the division of the fault injection process into campaigns and the division of a campaign into studies is left to the discretion of the user. However, it is desirable that a study consist of a set of correlated fault injections and a campaign consist of all the related studies, since combining results across different campaigns is not possible while obtaining measures in Loki. Each study consists of a set of *experiments*, each of which is one run of the distributed application along with the fault injections corresponding to the study. Thus, an experiment can be thought of as an instance of its study, with the state machine following the state machine specification, and fault injections being performed according to the fault specification. Multiple experiments (instances) are conducted for each study to obtain results that are accurate.

## 2.3 Evaluation of a System Using Loki

After the campaigns have been specified, the actual evaluation of the system has to be performed. Figure 2.1 illustrates the process of evaluating a system using Loki. Procedurally, the process can be divided into three main phases, namely the runtime phase, the analysis phase, and the measure estimation phase. There are also synchronization-message-passing mini-phases before and after each experiment in the runtime phase. Note that each of these three phases is executed for each experiment within every study, in each campaign. The following sections describe these phases in detail.

## 2.4 The Runtime Phase

The runtime phase is the phase during which fault injections are performed during experiment runs. It involves running the distributed system, injecting faults into the system at appropriate times, collecting observations, and recording them. To do this, the runtime phase makes use of the Loki runtime. During the running of an experiment, the Loki runtime

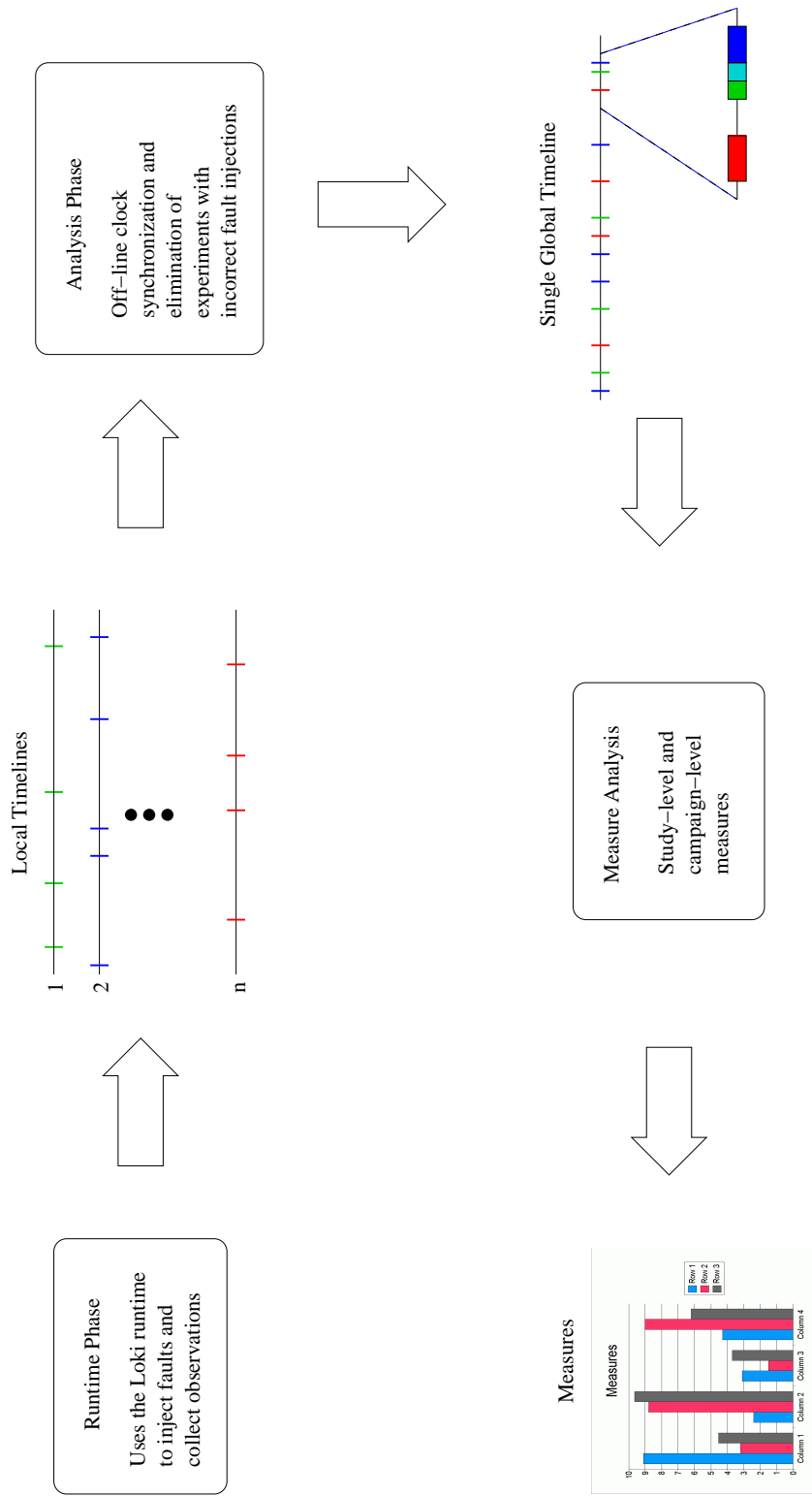


Figure 2.1: Evaluation of a System using Loki

attached to each node performs the above functions on that node.

The Loki runtime interacts with the system under study using the system-dependent probe. The probe has to be defined by the system designer as part of the application instrumentation. The probe notifies the node's runtime of local events occurring in the node's component. The fault injections into the component are also performed by the probe.

The state machine specifications are used in maintaining the partial view of the global state, and the fault specifications are used in triggering fault injections. The state machine tracks the local state of the component using the state machine specification and the probe's local event notifications. The state machine transport notifies the remote state machines of local state changes and also receives state change notifications from the remote state machines. The local state is used along with the remote state change notifications to maintain the required partial view of global state. Whenever the partial view of global state changes, the node's fault parser evaluates all the Boolean fault expressions given in the fault specification. If a fault expression that was previously false now transitions to true, the parser instructs the probe to inject the corresponding fault, and the probe then does the actual fault injection.

During the course of an experiment, the recorder records the times at which local state changes and fault injections occur, into a local timeline. A detailed explanation of the Loki runtime and the functions of its various components are given in Chapter 3. Note that the runtime uses only the user-specified state change notifications between nodes to keep track of the partial view of the global state. Also, a LAN separate from the one used by the system can be used for transmitting the notifications. Additionally, to be as non-intrusive to the system as possible, the runtime does not block the system while these notifications are in transit. That means that the system could change state while the notification is in transit; this implies that the partial view could sometimes be out-of-date. That could lead to incorrect fault injections in some experiments. If we did not remove these experiments as described in the next section, this could lead to incorrect measures.

## 2.5 Analysis Phase

The analysis phase prevents the errors that are mentioned in the previous section by conducting a post-runtime check on every fault injection to determine whether it has indeed been performed in the desired state. An off-line check is used to avoid the expense and intrusiveness of an on-line check. The results of the incorrect fault injections are discarded, and only the correct fault injections are used in computing the measures. The post-runtime check involves placing the local timelines from each of the state machines into a single global timeline, and then using the fault specifications to determine whether each fault was injected in the right state. This process is explained in detail below.

In particular, Loki uses an off-line clock synchronization algorithm to calibrate the clocks on the multiple machines on which the fault injector operates, so that all the local timelines of an experiment can be combined into a single global timeline. One machine's clock is taken as the reference, and the offset and drift rate of other machines' clocks are estimated relative to the reference clock. These offsets and drift rates are then used to place all the times onto a single global timeline.

Methods used to calibrate clocks in distributed systems include hardware methods, software methods, or a combination of the two. The choice of a method involves a trade-off among accuracy, intrusiveness, and portability. Hardware methods are highly accurate and not very intrusive, but they are generally system-specific. Conversely, software methods increase the portability, but decrease accuracy and increase intrusiveness. Finally, the hardware in the methods combining hardware and software reduces the intrusion and improves accuracy compared to the pure software methods. Since portability is an important concern in Loki, and the accuracy provided by software-based methods is sufficient, a software-based approach is used in Loki. However, the time is read from the hardware clock of the processor (whenever possible) to improve the accuracy of measurement.

The analysis phase assumes that the drifts of the processor clocks of the different machines

in the distributed system are linear. Therefore, if there are  $m$  machines in the system, numbered 1 to  $m$ , we have the following relation between the processor clock time  $C_i(t)$  on machine  $i$  and the processor clock time  $C_j(t)$  on machine  $j$ , where  $t$  is the physical clock time:

$$C_j(t) \approx \alpha_{ij} + \beta_{ij}C_i(t), \quad i, j = 1, \dots, m \quad (2.1)$$

where  $\alpha_{ij}$  is the offset between the clocks of machine  $i$  and machine  $j$  at  $t = 0$ , and  $\beta_{ij}$  is the drift of the clock of machine  $j$  with respect to the clock of machine  $i$ .

If a machine  $r$  is chosen as the reference machine, the calibration of the clocks of the machines in the system is reduced to a computation of  $\alpha_{ri}$  and  $\beta_{ri}$ , for  $i = 1, \dots, m$ . (It can be easily seen that  $\alpha_{rr} = 0$  and  $\beta_{rr} = 1$ .) The method used in Loki to compute these values is described in detail in [9]. It involves passing synchronization messages in two synchronization-message-passing mini-phases before and after each experiment during the runtime phase, and using a convex hull algorithm to compute the required values. Note that the synchronization messages are passed before and after each experiment and not during the experiment, so that the intrusiveness into application execution is reduced. Furthermore, note that the algorithm does not compute exact values for  $\alpha_{ri}$  and  $\beta_{ri}$ . Instead, it computes the lower and upper bounds,  $\alpha_{ri}^-$  and  $\alpha_{ri}^+$ , and  $\beta_{ri}^-$  and  $\beta_{ri}^+$ , respectively. Unlike confidence intervals for which a value has a high probability of being in a certain interval, the correct values of  $\alpha_{ri}$  and  $\beta_{ri}$  are always in the intervals  $[\alpha_{ri}^-, \alpha_{ri}^+]$  and  $[\beta_{ri}^-, \beta_{ri}^+]$  respectively (even though their exact values are unknown). It has been shown in [9] that this algorithm works well in practice and yields bounds on estimates of  $\alpha_{ri}$  and  $\beta_{ri}$  that are acceptably small.

As mentioned above, every local state change and fault injection in a node are recorded in its local timeline. The times used in the local timeline are the local times of the machine. During the conversion of the local timelines of all the nodes into a single global timeline, the occurrence times of all the events and fault injections have to be projected into a single

(reference) timeline. Suppose an event occurred on machine  $i$  at physical clock time  $T$  (i.e., local time on machine  $i$  is  $C_i(T)$ ). Then, from Eqn. (2.1), we have the reference clock time as

$$C_r(T) = \frac{C_i(T) - \alpha_{ri}}{\beta_{ri}}$$

However, since only the bounds for  $\alpha_{ri}$  and  $\beta_{ri}$  are known, only the upper and lower bounds of  $C_r(T)$  can be found. This is done as follows:

$$\begin{aligned} C_r(T)^- &= \frac{C_i(T) - \alpha_{ri}^+}{\beta_{ri}^+} \\ C_r(T)^+ &= \frac{C_i(T) - \alpha_{ri}^-}{\beta_{ri}^-} \end{aligned}$$

Therefore, an event occurring at time  $T$  on the physical clock time on machine  $i$  corresponds to an event occurring between bounds  $C_r(T)^-$  and  $C_r(T)^+$  on the reference machine  $r$ . Using this method, the events in the local timeline of all the nodes can be projected onto a (reference) global timeline. In practice, the difference between the bounds  $C_r(T)^-$  and  $C_r(T)^+$  has been found to be quite small if all the machines in the system are on a LAN (i.e., average message delay between the machines is small) [9].

After the conversion to the global timeline, all the fault injections are checked to determine whether they were proper, i.e., they have occurred in the correct global state as specified in the fault specification. This is done with a check that sees whether the time interval between the upper and lower global-time bounds of a fault injection completely lies within the time interval between the upper and lower global-time bounds of the correct global state. More specifically, the upper bound of the state start time and lower bound of the fault injection time are used to determine whether the fault was injected after the state was entered. Likewise, the lower bound of the state end time and upper bound of the fault injection time are used to determine whether the fault was injected before the state

was exited. If both the criteria are met, the fault was injected as intended. Note that even if both criteria are not met, it may be the case that the fault was injected correctly, but Loki conservatively assumes that it was not, to be sure that no experiments with incorrect fault injections are mistakenly deemed to be correct. This procedure is repeated for each injection that should have been made in the experiment; the experiment is only marked as successful if all the injections in the experiment were done correctly. If any of the fault injections were done incorrectly, the experiment results are discarded and are not used for measure computation.

## 2.6 Measure Estimation Phase

By the end of the analysis phase, the results of all the experiments with incorrect fault injections have been discarded, and only the results of the experiments with correct fault injections have been retained. The next phase in Loki, namely the measure estimation phase, allows the user to obtain measures from the results of the correct fault injections. He or she can use these measures to assess the dependability and performance of his/her distributed application. For this purpose, Loki contains a flexible language for specifying a wide range of dependability and performance measures. Also, Loki uses several statistical features to estimate these measures with high accuracy. In Loki, measures are defined at two levels: the study level and the campaign level. A measure at the study level consists of an ordered sequence of (subset selection, predicate, observation function) triples, and is associated with all the experiments in a study. Once these sequences have been defined for all studies, measures are defined across studies using one of the following two approaches. The first one, called *simple sampling measure*, considers the experiment results of all the studies to be similar, i.e., to be instances of the same random variable. The second approach, called “stratified sampling,” considers the experiment results of each study as a separate random variable. These random variables are then combined to get a campaign measure.

If the function used to combine the random variables is a linearly weighted function, the obtained campaign measure is a *stratified weighted measure*. If it is a user-defined function, the obtained measure is a *stratified user measure*. Chapter 4 explains the measure estimation process in Loki in greater detail.

# Chapter 3

## Loki Runtime Architecture

### 3.1 Introduction

The Loki runtime manages the runtime phase of the fault injection process. During every experiment execution, it maintains the partial view of the global state for each of the components of the distributed system. It also performs fault injections and collects observations when necessary. This chapter describes the architecture of the Loki runtime in detail. More specifically, the original Loki runtime and its shortcomings are first detailed. Then the desirable features of an enhanced runtime, which overcomes the shortcomings of the original runtime, are presented. The different designs that could have been used for the enhanced architecture are then identified and described. Based on its advantages, one of the designs is chosen for implementation of the enhanced runtime and is explained in detail.

### 3.2 The Previous Loki Runtime

#### 3.2.1 Overview of the Previous Runtime

The original Loki runtime is shown in Figure 3.1. The original version of the runtime provides the core functionality required for fault injection in Loki. It consists of a state machine, state machine transport, fault parser, recorder, and probe for each node in the

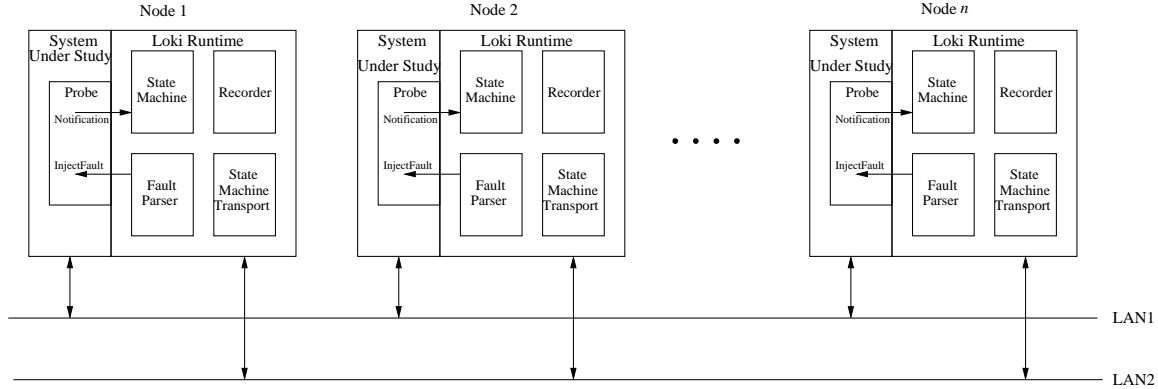


Figure 3.1: Original Loki Runtime Architecture

system. The functionalities of these components (except the state machine transport) are similar to those of the corresponding components in the new runtime, and are described in detail in Section 3.5. Using local event notifications and remote state notifications, the original runtime maintains the partial view of the global state of the distributed system that is necessary for fault injection. When the system transitions to the desired states, it instructs the probe to perform fault injections. It uses Boolean fault expressions and provides the user with considerable freedom in his/her choice of fault types. The runtime also collects observations regarding state changes and fault injections that are used for off-line analysis and measure computation. A detailed description of the original Loki runtime architecture can be found in [10].

### 3.2.2 Performance Analysis

The main goal in designing the original version of the runtime was to prove the concepts underlying Loki. The secondary goal was to obtain an efficient runtime implementation with very low intrusion and high efficiency in fault injection and measurement. To verify that these goals were met, a performance analysis of the original Loki runtime was conducted, the detailed results of which are presented in [10]. A simple test application was used during the performance analysis, and the efficiency of Loki in injecting faults was measured.

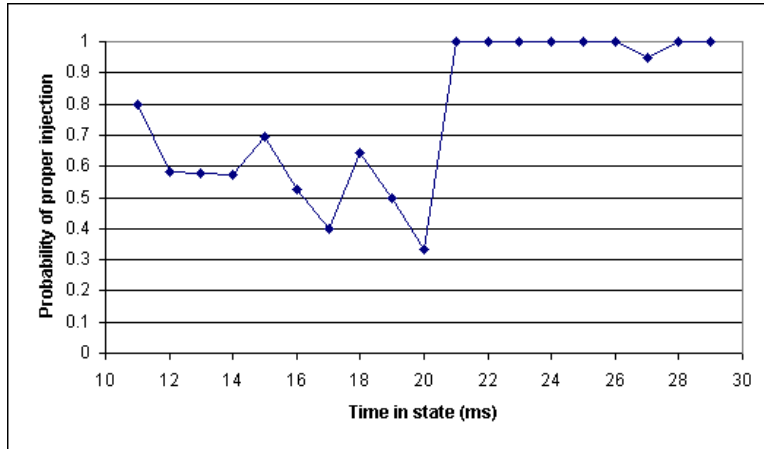


Figure 3.2: Correct Fault Injection Probability as a Function of Time Spent in a State (10ms Linux Timeslice)

This involved varying the amount of time the application spent in a particular global state in which a fault was to be injected, and measuring the percentage of time the fault was correctly injected by Loki. This measurement was done for two values (10ms and 1ms) of the underlying OS's (Linux's) timeslice, to determine the effect of the OS on Loki's performance. Figures 3.2 and 3.3 show the measurement results when the OS timeslice is 10ms and 1ms respectively. It can be easily seen that the original Loki runtime was able to inject faults in the desired global states if the application stayed in the state for a time greater than a couple of OS timeslices. This shows that the actual time taken by a notification message on the network, and the overhead incurred due to the fault injection by Loki, are minimal compared to the OS context switching overhead incurred during the sending and receiving of a notification message. It also shows that the accuracy of fault injection can be further improved if the OS overhead in sending and receiving messages was decreased. Thus, the original runtime was an accurate and minimally intrusive runtime implementation that proved Loki's basic concepts.

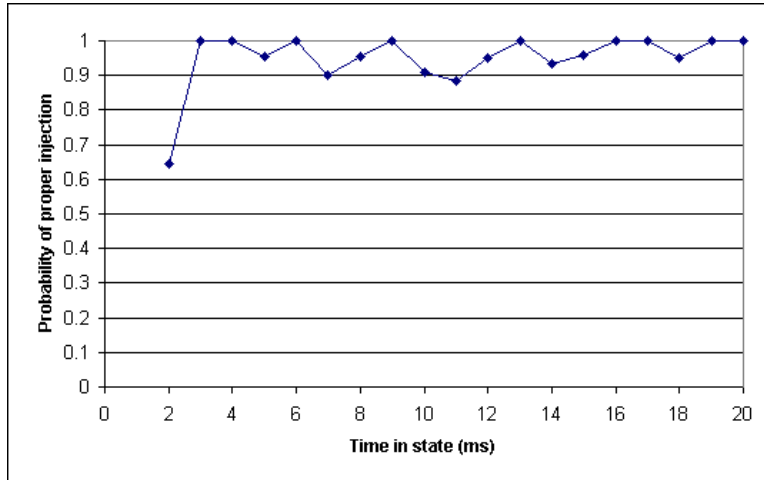


Figure 3.3: Correct Fault Injection Probability as a Function of Time Spent in a State (1ms Linux Timeslice)

### 3.3 Shortcomings of the Original Runtime

Despite its advantages and accuracy, a major shortcoming of the original runtime is that it is static in nature, i.e., it does not allow nodes to exit from or enter into the runtime system dynamically. This means that nodes can be started only at the beginning of the experiment and can exit only at the end of the experiment. However, that limitation is unacceptable, because processes can crash and restart during the fault injection of a general distributed system. Another minor point of note is that in this runtime, state machines in the same host communicate using TCP/IP that could be optimized using some form of interprocess communication (IPC), such as shared memory or pipes.

### 3.4 Design Choices for the Enhanced Loki Runtime

As mentioned in the previous section, it was necessary for the enhanced runtime to support dynamic entry and exit of nodes. Also, the new runtime needed to maintain the low intrusion and high efficiency of the previous runtime, both during normal execution and during node entry and exit. Note that to achieve the dynamic entry and exit of nodes, the design of

most of the core Loki runtime components (i.e., state machine, fault parser, recorder, and probe) and their interaction did not need to be changed. Only the transport mechanism needed to be redesigned so that a dynamically entering node's state machine can establish communication with all the existing state machines in the system. Consequently, the new transport mechanism should have low delays during normal notification message delivery (as in the previous transport mechanism), and also have low overhead and low intrusion during the entry and exit of nodes.

### 3.4.1 The Design Choices

Three possible high-level designs were identified for the enhanced Loki runtime architecture. They are shown in Figure 3.4. These designs use *daemons* to provide the functionality needed by the enhanced runtime. A daemon is a process that always executes in the system, independent of the entry or exit of nodes, and monitors all the nodes associated with it. The distinction between the three designs is in the number of and organization of daemons in the system and in the number of nodes associated with each of the daemons. In the *centralized design*, there is a single global daemon that caters to all the nodes in the system through TCP/IP links. The *partially distributed design* has one daemon per host machine; this daemon caters to all the nodes on that host using IPC connections (such as shared memory). In the *fully distributed design*, there is one daemon per node, connected to the node by IPC. In both of the distributed designs, the daemons themselves are connected to each other using TCP/IP.

In each of the three designs, communication between the nodes' state machines can be done in one of two ways: either the state machines communicate directly with each other, or they communicate via the daemons. If they communicate directly, the nodes will have direct TCP/IP connections in addition to the connections with the daemons. In all of the above designs, when a node crashes, its corresponding daemon detects the crash (because of the breaking of the communication link), and writes the crash event information to the local

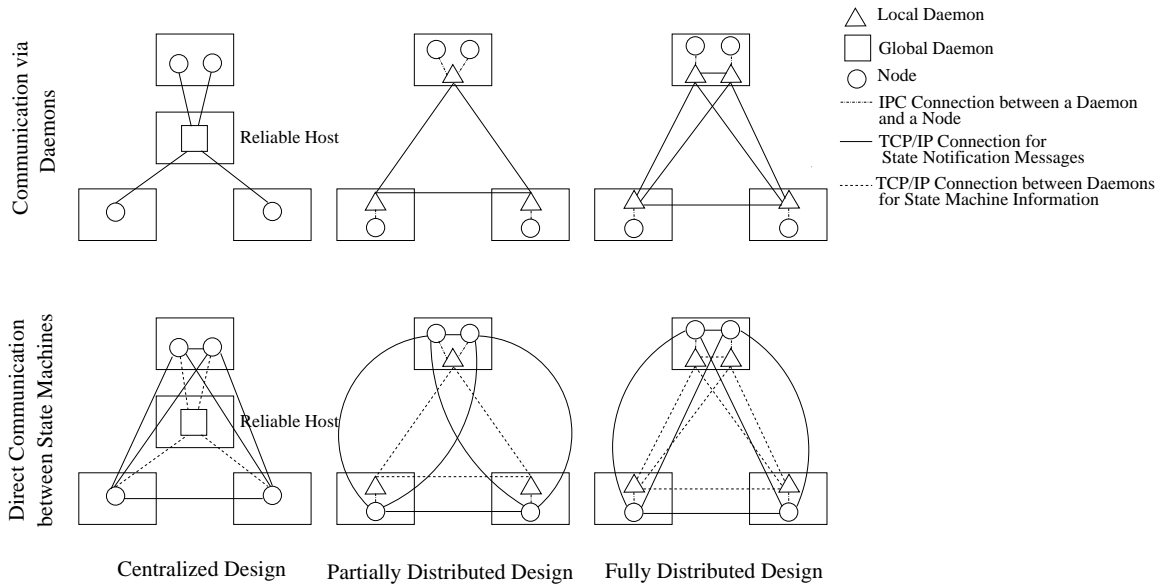


Figure 3.4: Design Choices for the Loki Runtime Architecture

timeline of the node’s state machine. When a node restarts, it uses its daemon to establish communication with all the other state machines in the system. If the communication between the state machines is direct, the new node obtains information about all the state machines in the system from the daemon and establishes TCP/IP connections with all of them. Otherwise, it establishes a connection only with its daemon and communicates through it.

### 3.4.2 Comparison of the Design Choices

The advantage of the centralized design, irrespective of whether the communication is through the daemon, is that it is completely dynamic, in that new hosts can be added to the system at runtime. The disadvantage is that there could be a relatively long delay in detecting a node crash, since the detection is based on the breaking of a TCP/IP connection. If the communication is through the daemon, an extra advantage of this design is the low overhead and low intrusion on entry and exit of a node, since the node’s state machine needs to connect to and disconnect from only the global daemon. There are also disadvantages, in

that notification messages between state machines are slow since each message requires two hops, and the system has limited scalability, since the global daemon could become a bottleneck as the number of nodes increases. If the state machines communicate directly with each other, there is an advantage in that notification messages between state machines are faster. However, there is also a disadvantage in that on a node entry or exit, the node's state machine has to make or break TCP/IP connections with all the other state machines. As the number of nodes increases, the overhead and intrusion of these operations could become large.

Note that in the partially distributed design, the set of hosts in the system and hence the location of local daemons on the hosts should be known prior to experiment execution. Hence, independent of the communication mechanism, the disadvantage of this design is that the set of hosts in the system is static, i.e., new hosts cannot be added to the system during experiment execution. If communication is through the daemons, the advantages of the partially distributed design are that there are low overhead and low intrusion on entry and exit of nodes, multicast of notifications to state machines is more efficient, and notifications between state machines on the same host go through IPC and hence are more efficient. However, there is an added disadvantage, in that there is a slight increase in the delay of notification messages between state machines on different hosts, since each message involves two IPC communications and one TCP/IP communication. If the state machines directly communicate with each other, the advantage is that the notification messages are faster, but the disadvantage is the high overhead and intrusion on a node entry or exit, since the node's state machine has to make or break TCP/IP connections with all the other state machines.

In the fully distributed design, the set of nodes must be known prior to experiment execution. As a result, irrespective of the communication mechanism, the disadvantage of this design is that the set of the nodes in the system is static, i.e., the number and location of state machines in the system is static. If the communication is through the daemon, then

there is lower overhead and low intrusion during node entry and exit, and higher overhead during notification messages than if the state machines communicate directly with each other.

The main goal of the enhanced architecture is to provide dynamic entry and exit of nodes such that a node that crashed on one host can restart on another host. Most reliable distributed systems allow crashed processes to restart on different hosts, so this capability is essential. However, the fully distributed design has a static list of nodes and hence only supports restarts on the same host. Since this would be very restrictive, the fully distributed design is not suitable for the enhanced Loki runtime architecture.

In the centralized design, there could be a delay before the global daemon detects a node crash. This would cause an incorrect time for the crash event to be recorded in the local timeline. Also, there is no method of finding out the magnitude of error in this incorrect time. This could lead to incorrect analysis and hence incorrect measure estimations. Additionally, the centralized design limits scalability both when the communication is through the daemon and when the state machines communicate directly with each other. Hence, the centralized design is not acceptable.

The static list of hosts in the partially distributed design is not a limitation in practice, since all the hosts in a distributed system are generally known before experiment execution. Comparing the two communication mechanisms in the partially distributed design, we find that communication through daemons is more efficient than direct communication between state machines. The communication using daemons offers more efficient multicast of notifications, more efficient notifications between state machines on the same host, and lower overhead on node entry and exit, as compared to direct communication between state machines<sup>1</sup>. Also, in communication using daemons, if the runtime portion of a node becomes

---

<sup>1</sup>Assuming that the rate of processing of messages by a host is faster than the rate at which the network delivers them, it might be argued that if the number of nodes on a host becomes large, the local daemon for the host would become a communication bottleneck, since all the communication is queued at the daemon. However, this is not the case, since all of the host's communication is queued at the network interface of the host anyway.

corrupted during fault injection, proper checks implemented at the local daemon could help contain the effect of the fault. These checks could verify the validity of the notification messages being sent out by a node before transmitting them to the remote node. Furthermore, since in current systems the IPC delay is on the order of  $20\mu s$  and the TCP/IP delay is on the order of  $150\mu s$ , we see that the overhead for notification messages in communication using daemons is not dramatically larger than in direct communication between state machines. Therefore, the design of choice for the new runtime is the partially distributed design with communication through daemons.

## 3.5 Architecture of the New Loki Runtime

Figure 3.5 illustrates the new Loki runtime. It shows an example of a system with a runtime that has four nodes on three hosts. Note that in addition to the local daemons of the partially distributed design, there is also a central daemon to which all the local daemons are connected. Each node of the runtime consists of the system under study along with the state machine, state machine transport, fault parser, recorder, and probe. The state machine, state machine transport, fault parser, and recorder are independent of the system under study, while the probe is highly dependent on it. These components of the runtime are described in detail in the following sections.

### 3.5.1 Central Daemon

There are two types of daemons in the Loki runtime: a single central daemon, and one local daemon for each host in the distributed system. The central daemon is responsible for the overall management of each experiment in the runtime phase. At the beginning of every experiment, it starts up the local daemons and also instructs them to start up the state machines specified by the user. The user can specify the state machines to be started at the beginning of an experiment using the node file. The node file has entries of the following

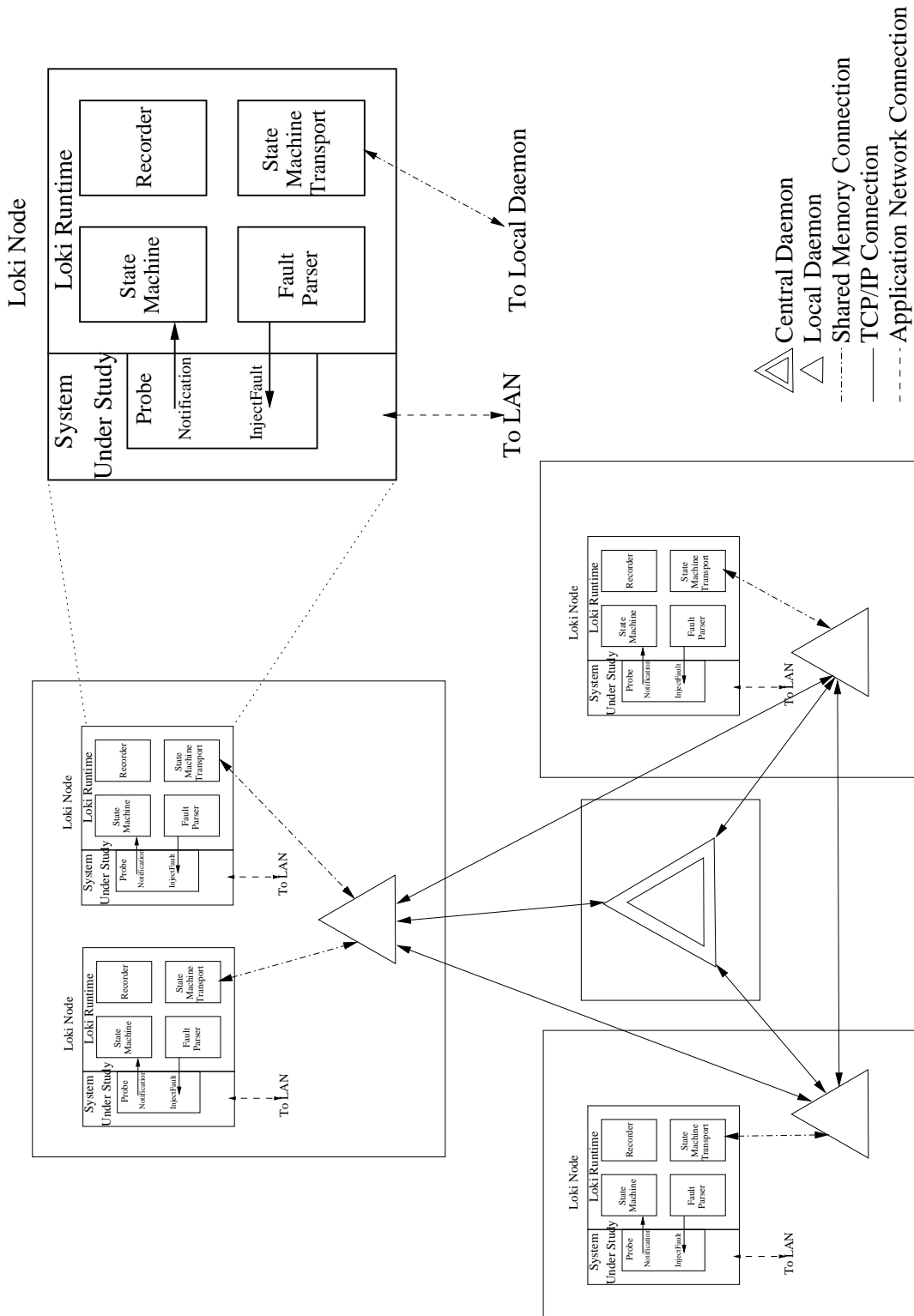


Figure 3.5: The New Loki Runtime Architecture

format, one per line, for each state machine:

```
<SM NickName> [<HostName>]
```

If the optional `<HostName>` field is present, then at the beginning of every experiment, the central daemon instructs the local daemon on `<HostName>` to start the state machine `<SM NickName>` on that host. Otherwise, the state machine is not started at the beginning of an experiment. During the execution of an experiment, the central daemon checks for any abnormalities in the system execution, such as a local daemon crash. If an abnormality occurs, the central daemon instructs the local daemons to kill all the state machines, and aborts the experiment. Also, if an experiment takes longer than the timeout value specified by the user, the central daemon considers the experiment as hung, instructs the local daemons to kill all the state machines, and aborts the experiment.

When an experiment completes, the central daemon receives notification messages indicating the experiment completion from all the local daemons. At this point it considers the experiment complete and starts the next experiment. Currently, the central daemon is integrated into the Loki graphical user interface and is described in greater detail in [11]. In the future, the central daemon may also take care of a host crash and reboot during the execution of an experiment, as explained in Section 3.6.4.

### 3.5.2 Local Daemon

In Loki, there is one local daemon per host in the system, and it is connected to all the other local daemons using TCP/IP links. The main functions of the local daemon are to take care of entry, exit, crash, and restart of state machines, to provide communication between state machines, to start and kill state machines based on the instructions of the central daemon, and to check for experiment completion. These functions are explained below in greater detail.

The local daemons, on being started by the central daemon, contact each other at the user-specified ports. To do this, they make use of the daemon startup file, which has to be specified by the user before the start of the experiments. The daemon startup file has one entry per line of the following format, indicating the corresponding port for the local daemon on each host.

```
<HostName> <PortNumber>
```

After the local daemons connect to each other using TCP/IP, they create a shared memory region and an associated semaphore and write the shared memory and semaphore information, which are used by the state machines to contact the local daemons, into a daemon contact file. The state machines on startup and on restart use this information to contact the local daemons on their respective hosts. The format of each line of the daemon contact file is as follows. It gives the shared memory and semaphore identifiers needed to contact the local daemon on each host.

```
<HostName> <SharedMemoryID> <SemaphoreID>
```

Then the local daemons wait for connections from new and restarted state machines. When a new state machine starts up, it contacts its corresponding local daemon, and the local daemon spawns a separate thread to service the state machine's notification messages. Then two new pairs of shared memories and semaphores are created for communication between the service thread and the state machine. The local daemon also obtains the state machine's process id and local timeline file name. The local daemon also acts as a watchdog for all of its associated state machines. When a state machine crashes, the local daemon detects the crash either due to the deletion of the shared memories and semaphores by the state machine when it crashed, or because the state machine does not respond to the

watchdog messages. On a state machine crash, the local daemon writes a crash event and crash state into the local timeline of the state machine, and notifies all the other local daemons of the crash. The interaction of the local daemon with a restarted state machine is similar to its interaction with a new one.

When a state machine wants to send a notification message to a remote state machine, the state machine transport of the state machine sends the message to its service thread in its local daemon. The service thread then sends the message to the local daemon of the remote state machine using TCP/IP, which in turn passes it on to the remote state machine's state machine transport. When the central daemon wants to start a state machine on a particular host, it instructs the local daemon on that host to perform this action. When the central daemon wants to kill a state machine, it instructs its corresponding local daemon to do so, which then uses the process id of the state machine to kill it. The local daemons also check for experiment completion on every exit or crash of a state machine. An experiment is deemed complete if there are no state machines executing in the system, i.e., all the state machines have either exited or crashed. On the completion of an experiment, the local daemons notify the central daemon of the experiment completion.

### **3.5.3 State Machine**

A state machine keeps track of the partial view of global state necessary to inject faults in the corresponding node's application. Even though multiple nodes use the same executable, there is one state machine per node, and it has to be given a unique name. For example, during the fault injection of a replication scheme, each replica constitutes one node with a unique name. The tracking of the partial view of global state includes tracking the local state of the node's application as well as maintaining the state of remote state machines that is needed for fault injection. To keep track of the local state, the state machine uses the state machine specification file provided by the user and the local event notifications sent by the probe. The state machine specification file indicates all the states in which the

state machine can be, along with the transitions between them and the events that cause these transitions, i.e., it specifies the application's execution as a state machine at the level of abstraction needed for fault injection. There is one state machine specification file for each state machine. The state machine specification for a particular state machine has the following format:

```
global_state_list
<list_of_states>
end_global_state_list
event_list
<list_of_events>
end_event_list
state <state_1> [notify <nickname_1_1>, ... <nickname_1_i>]
<event_1> <next_state_1_1>
:
<event_m> <next_state_1_m>
...
state <state_n> [notify <nickname_n_1>, ... <nickname_n_i>]
<event_1> <next_state_n_1>
:
<event_m> <next_state_n_m>
```

The `list_of_states` contains a list of the global states of all the state machines in the system, with one state specified per line. The `list_of_events` consists of the list of local events in this particular state machine, with one event specified per line. After the event list, the specification for each state is given. For each state, the list of state machines to be notified when this state machine enters the state is specified after the `notify` keyword. The

new states to which the state machine transitions when an event occurs in the state are then specified.

When a local event occurs in the node's application, the state machine is notified of it by the probe. The state machine then uses its current state, the event, and the state machine specification to determine the new state and then transition to it. Also, it sends state notifications (using the state machine transport) to any other state machines that are to be notified of the state machine's new state, as given after the `notify` keyword of the state definition of the new state. The state machines use these state notifications to keep track of the necessary state of the remote state machines. Furthermore, on every change in its partial view of global state, the state machine notifies the fault parser.

### **3.5.4 State Machine Transport**

The state machines make use of their respective state machine transports to send notification messages to each other. When a node starts up, its state machine transport looks up the daemon contact file and establishes a connection with the local daemon on its host. It can then send notifications to and receive notifications from the state machine transports of other state machines using the intermediate local daemons. When the state machine requests that the state machine transport send a notification to a remote state machine, the state machine transport adds the necessary headers and sends the notification to the remote state machine transport through its own local daemon and the local daemon of the remote state machine. When the state machine transport receives a notification for its state machine, it forwards it to the state machine.

### **3.5.5 Fault Parser**

On every change in the partial view of global state, the state machine notifies its corresponding fault parser. The fault parser then checks whether a fault needs to be injected in that

global state. It makes use of the fault specification provided by the user, and the partial view of global state, to perform the check. Each entry of the fault specification is of the following format:

```
<FaultName> <BooleanFaultExpression> <once|always>
```

If the value of the `<BooleanFaultExpression>` transitions from false to true because of a global state change, then the fault `<FaultName>` has to be injected, conditional upon the `<once|always>` field. The `<once|always>` field indicates whether the fault is to be injected the first time the state is entered (once) or whenever the state is entered (always), from a different global state. The `<BooleanFaultExpression>` consists of entries of the form `(State Machine:State)` combined using the AND (`&`), OR (`|`), and NOT (`~`) operators. For example,

```
F1 ((SM1:ELECT) & (SM2:FOLLOW)) always
```

specifies that the fault F1 has to be injected whenever (**always**) the system enters the global state in which the Boolean expression `((SM1:ELECT) & (SM2:FOLLOW))` is satisfied (i.e., the state machine SM1 is in the state ELECT and state machine SM2 is in the state FOLLOW) from a global state in which it is not satisfied.

So, on every state change, the fault parser parses all the Boolean fault expressions and determines whether any of them have transitioned from false to true as a result of the state change. For each of the fault expressions that has transitioned from false to true, it checks whether the `<once|always>` field allows for fault injection; if it does, the parser instructs the probe to inject the fault.

### 3.5.6 Recorder

The recorder is the component of the Loki runtime that records the relevant data into a local timeline file. The relevant data includes the information regarding local state changes and fault injections along with their times of occurrence, and also any messages that the user would want to include in the local timeline. The format of the local timeline file is as follows:

```
<mySMnickName>
state_machine_list
<index 1> <SMNickName 1>
...
<index n> <SMNickName n>
end_state_machine_list
global_state_list
<index 1> <stateName 1>
...
<index m> <stateName m>
end_global_state_list
event_list
<index 1> <eventName 1>
...
<index k> <eventName k>
end_event_list
fault_list
<index 1> <faultName 1> <faultExpr 1> <once|always>
...
<index i> <faultName i> <faultExpr i> <once|always>
```

```
end_fault_list
local_timeline
<recorded local timeline events>
end_local_timeline
```

In the above format, `mySMNickName` is the nickname of the state machine corresponding to the local timeline. The `state_machine_list` contains one line for each state machine; the line contains the index of the state machine and the state machine nickname. The `global_state_list` contains the list of states in all the state machines and their corresponding indices. The `event_list` contains the list of all the local events in the state machine corresponding to the local timeline, along with their indices. The `fault_list` contains the fault specification for the state machine along with an index for each fault. The state machine, state, event, and fault indices are used in the local timeline events in place of the corresponding names. This makes the local timeline compact and decreases intrusion during recording of the local timeline. Each of the recorded local timeline events can be either a state change or a fault injection. The format for a state change is as follows:

```
STATE_CHANGE <EventIndex> <NewStateIndex> <EventTime.Hi> <EventTime.Lo>
```

where `<EventTime.Hi>` is the upper 32 bits of the 64-bit event time and `<EventTime.Lo>` is the lower 32 bits. The format for a fault injection is as follows:

```
FAULT_INJECTION <FaultIndex> <FaultInjectionTime.Hi> <FaultInjectionTime.Lo>
```

The fault injection time is also a 64-bit number and the `<FaultInjectionTime.Hi>` and `<FaultInjectionTime.Lo>` correspond to its upper and lower 32 bits. `STATE_CHANGE` and `FAULT_INJECTION` are numerical constants with values 0 and 1 respectively.

### 3.5.7 Probe

The probe is the system-dependent part of the Loki runtime. (Note that *system-dependent* means that the user has to write the code for the probe implementation. Therefore, the state machine is system-independent, even though the state machine specification is system-dependent, since the state machine code is not written by the user.) The user should implement the probe while he/she is instrumenting the system under study. The two functions of the probe are to notify the state machine of any local events occurring in the application, and to perform the actual fault injection when instructed to do so by the fault parser. The first thing to be done during the instrumentation process is to rename the `main()` function of the application to `appMain()`. However, the arguments of the `main()` function need no modification.

To notify the state machine of any local events, the probe makes use of the `notifyEvent()` method of the state machine and sends the event name and the time of its occurrence to the state machine. However, the first event notification that the probe sends is considered as a state and is used to initialize the state of the state machine. Note that since the state machine uses the state machine specification to track the partial view of global state, the notifications that are sent by the probe should be consistent with the state machine specification. The probe should also implement the `injectFault()` method, which performs the actual fault injection and returns the time of injection. Whenever the fault parser determines that a fault is to be injected, it calls the `injectFault()` method of the probe along with the fault name. The injected fault could cause a signal and a subsequent crash of the node. If the user's code has overridden the signal handler, then it should call the `notifyOnCrash()` method of the state machine before it exits the process. This is done so that the local daemon corresponding to the node registers the crash. When the state machine exits cleanly, it should notify its local daemon using the `notifyOnExit()` method. Otherwise, the watchdog functionality of the local daemon would consider the state machine to have crashed. For an example of a

probe, refer to Section 5.5.

Note that though the interaction between the probe and the rest of the Loki runtime is by in-process method calls, the Loki probe could be designed to be very general, like a wrapper around a process, or a layer in a protocol stack. This could be achieved by having one part of the probe in the same process as the other components of the Loki runtime, and having another part be the wrapper around a process or a layer in the protocol stack. The two parts of the probe could interact with each other to obtain the desired result. Also, in a similar fashion, probes can be designed both when the source code of the application is available (by integrating the probe into the application), and when it is not (by making the probe a wrapper around the application). However, when the source code of the application is not available, the application has to provide a means for changing the arguments it passes to any node it starts. This has to be done since the Loki runtime attached to the started node needs arguments to identify its corresponding state machine.

Additionally, note that some state and event names are reserved in Loki. The reserved state names are BEGIN, EXIT, CRASH, and RESTART, and the reserved event names are CRASH, RESTART, and default.

## **3.6 Operation of the New Loki Runtime**

This section describes the operation of the new Loki runtime under different circumstances. Mores specifically, it first describes in detail the normal operation of the runtime. Then the sequence of steps performed on a node crash and restart are detailed. Finally, the behavior of the runtime on a host crash and reboot is described.

### **3.6.1 Normal Operation**

At the beginning of an experiment, the central daemon starts all the local daemons. The local daemons connect to each other and to the central daemon through TCP/IP. Then,

each of the local daemons creates a known shared memory region with semaphore and waits for connection requests from the local state machine transports. The local daemons store the identifiers of these known shared memory regions in a daemon contact file. The central daemon then instructs the corresponding local daemons to start up only those state machines that are specified by the user in the node file as state machines to be started at the beginning of an experiment. New nodes can enter the system or existing nodes can leave the system at any time during the experiment execution. When a node starts up, its state machine transport looks up the known shared memory region of the local daemon in the daemon contact file, and sends a connection request to it. The daemon, on servicing the request, creates a new shared memory region along with the associated semaphore for communication of notification messages. Each local daemon maintains the location of all the state machines. When a state machine transport sends the local daemon a state change notification along with a list of state machines to be notified, the daemon first looks up the local daemons of each of the recipient state machines and then forwards the notifications to them. These daemons in turn forward the notification to the state machine transports of the recipient state machines using shared memory. If there is a notification for a state machine that is currently not executing, the notification is discarded with a warning message. Note that the local daemon of the sending state machine needs to send only one notification per host, even if multiple state machines on the host are receiving it. Also, notifications between state machines on the same host go through shared memory and not through TCP/IP, and hence are more efficient. When there are no more nodes executing in the system (because all of them either crashed or exited), the experiment ends. At every state machine crash and exit, the local daemons perform a local check for experiment end. If the local check indicates that the experiment has ended, a local daemon sends a experiment end notification to the central daemon. When the central daemon receives experiment end notifications from all the local daemons, it considers that the experiment to have concluded, and then begins the next experiment run. To prevent an erroneous application from executing indefinitely, the user

can specify an application timeout value. If the experiment times out, the central daemon instructs the local daemons to kill all the state machines, and, after cleaning up the current experiment, starts the next experiment.

### **3.6.2 On a Node Crash**

When a node exits normally, the node's state machine sends an exit notification to all the other state machines. However, when the node crashes, the Loki runtime detects the crash in one of two ways. First, if a signal is generated due to the crash, the signal handler for the node deletes the shared memory region used to communicate with the local daemon and the associated semaphores. Because of this deletion, the local daemon is notified of the crash by the OS. It is assumed that the user has not overridden the signal handler. If, on the contrary, the signal handler has been overridden, then the user's code must call the `notifyOnCrash()` method of the state machine explicitly upon a node crash. Second, the local daemon functions as a watchdog and monitors all the state machines associated with it. If any of the state machines times out, it is assumed by the local daemon to have crashed. The user is given the flexibility to fix the timeout value. On detecting a crash, the local daemon writes the crash event to the local timeline of the crashed node's state machine and notifies all the other daemons of the crash.

### **3.6.3 On a Node Restart**

When a node crashes, the reliable distributed system could restart it, possibly on a different host. The new runtime provides support for this node restart. When a node is started, its state machine checks its local timeline to determine whether the node is a new one or a restarted one. (Note that the timeline file is NFS-mounted.) A restarted node's state machine writes restart event information to the local timeline. This information contains the name of the host on which the state machine was restarted, which is used during off-

line clock synchronization. Then the state machine connects to its local daemon much like a new state machine would. The local daemon sends notifications to all the other local daemons indicating that the state machine has restarted. The state machine then obtains state updates from all the other state machines to update its view of the global state. After that, the node executes like a normal node.

### **3.6.4 On a Host Crash and Reboot**

If a host crashes, the local daemon on the host also goes down. The central daemon and the other local daemons detect this because their TCP/IP connections with the crashed local daemon break. The central daemon waits for the host to boot back up and restarts the local daemon on it. The local daemon connects to the central daemon and all the other local daemons, and the experiment execution continues normally. This support for host crash and reboot has not yet been implemented in Loki.

## **3.7 Benefits of the Enhanced Runtime Architecture**

Several design choices for the new Loki runtime have been considered, and the partially distributed design with all communication through the daemons was chosen because of its advantages over the other design choices. The main advantage of the enhanced Loki runtime over the previous Loki runtime is that it allows dynamic entry and exit of nodes. Moreover, it has some additional advantages. It offers more efficient multicast of notification messages to state machines. Also, the notification messages between state machines on the same host go through shared memory and hence are more efficient than in the previous Loki runtime. Because of the hierarchical architecture, the new runtime is also more scalable, both during normal execution and during entry and exit of nodes.

## 3.8 Limitations of the Current Loki Implementation

Currently Loki has been implemented in C++ on Linux 2.2. Since the Loki library has to be linked with the application code and function calls should be possible from application code to Loki library and vice versa, the application has to be written in a language that provides this support. Additionally, in the current implementation, Loki has a few other limitations, which are listed below. These limitations are due to decisions made during implementation and are not inherent in the concepts underlying Loki. Future implementations might overcome these limitations.

- A network file system is currently needed for the proper functioning of Loki.
- The necessary state change notifications have to be manually specified by the user in the state machine specification file. They are not automatically deduced by Loki from the fault specifications.
- If the application starts up nodes on its own, it has to provide Loki with a way of specifying the arguments to the nodes. If the application does not provide this feature, changes in the application source code to implement this feature might be necessary.
- The application components cannot have arguments changing from one experiment run to another, since the arguments are written into the study files of the components' state machines.
- Every state machine that could possibly start during the execution of the system must be given a unique name and must be specified before the experiments are run.

# Chapter 4

## Measure Estimation in Loki

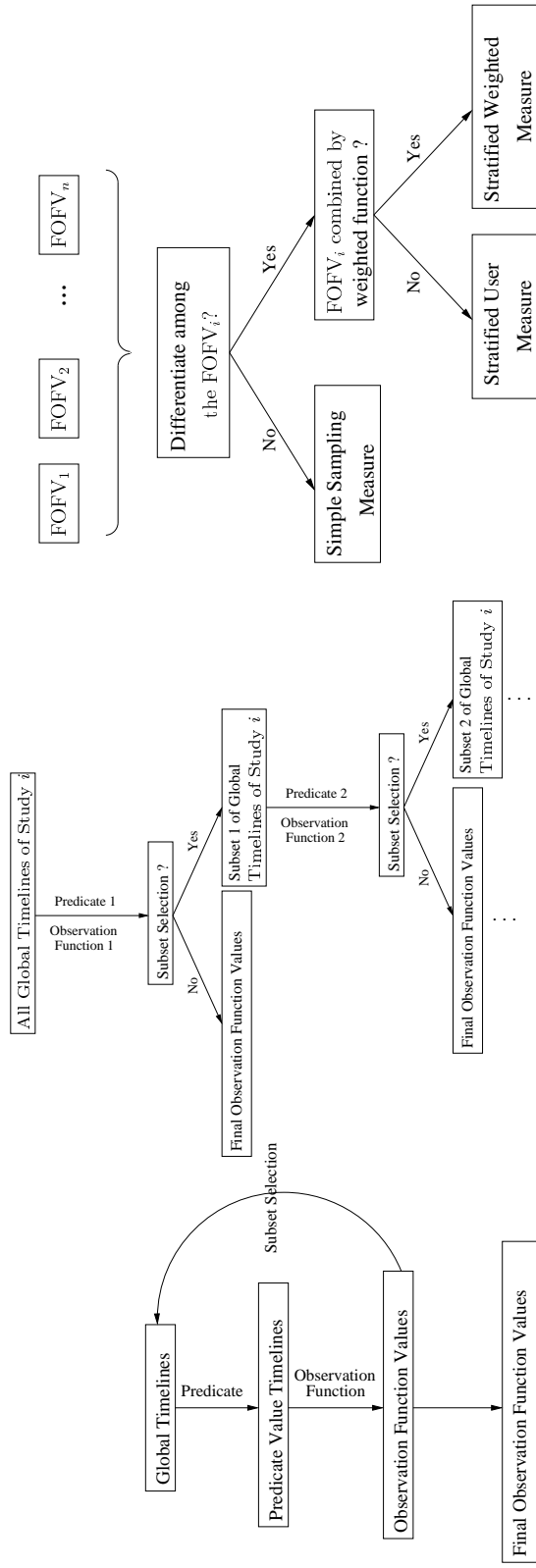
### 4.1 Introduction

Measure estimation is a key component of performance and dependability assessment using fault injection. It is important to note that the measures to be obtained are highly system- and user-dependent; for example, computing the system's coverage of a fault depends on the user's definition of a failure and recovery. Hence a fault injector should provide a means for the user to obtain a variety of accurate measures from the results of the fault injection experiments.

To this end, a flexible mechanism for measure estimation has been developed in Loki, which is the subject of this chapter. The chapter first describes the flexible measure language in Loki, which is used to specify a wide range of performance and dependability measures. Included are the descriptions of the levels at which measures are specified and the different types of measures at each level. The chapter then details the various statistical features used by Loki to compute the specified measures with high accuracy whenever possible.

### 4.2 Overview of Measure Estimation in Loki

This section briefly describes the method of specifying measures in Loki. The later sections describe this measure specification process, along with the terms used in this section, in



(c) Campaign-Level Measures (FOFV = Final Observation Function Value)

(b) Subset Selection in Study Level Measures

(a) Measures at the Study-Level

Figure 4.1: Measure Estimation in Loki

greater detail. Figure 4.1 shows the measure specification process in Loki.

As shown in Figure 4.1, measures in Loki are specified at two levels: the study level and the campaign level. A measure at the study level consists of an ordered sequence of (subset selection, predicate, observation function) triples, and is associated with all the experiments in a study. The output of applying a study measure to the global timeline of an experiment is the final observation function value for the experiment, if the experiment passes through all the subset selections successfully. Otherwise, the experiment is removed from further consideration in the measure estimation process involving this study measure. Once the study level measures have been specified for all studies, measures are defined across studies using one of two approaches. The first one, called the *simple sampling measure*, considers the experiment results of all the studies to be similar, i.e., to be instances of the same random variable. The second approach, called “stratified sampling,” considers the experiment results of each study to be a separate random variable. These random variables are then combined to get a campaign measure. If the function used to combine the random variables is a linearly weighted function, the obtained campaign measure is a *stratified weighted measure*. If it is a user-defined function, the obtained measure is a *stratified user measure*. Figure 4.1(c) shows these three types of campaign measures. A campaign measure takes the final observation function values of the study measure applied to all the experiments as input and gives as output, whenever possible, an accurate measure value.

The next few sections describe the measure estimation process in greater detail. More specifically, they describe the measures at the study level, including the concepts of predicate, predicate value timeline, observation function, observation function value, and subset selection. They also detail the campaign measures, along with the different types of campaign measures and the statistical computations performed to obtain accurate values for the measures.

## 4.3 Measures Defined at the Study Level

Measures at the study level are based on three concepts: *predicate*, *observation function*, and *subset selection*. Each measure is an ordered sequence of (subset selection, predicate, observation function) triples. These three concepts and the manner in which they are combined to obtain the study level measures are described in detail below.

### 4.3.1 Predicate

Predicates in Loki are used to query the global timeline (which was generated during the analysis phase described in Chapter 2) to identify whether certain conditions are satisfied. A predicate is a function that queries the different attributes of the state machines (i.e., states, events, and times), and is either true or false as a function of time.

Each *predicate* is an expression defined by tuples that are combined using AND, OR, and NOT operators. Each tuple queries a particular state machine for the occurrence of a state and/or event at specific times. Four different types of tuples can be defined in Loki: (state machine, state), (state machine, state, time), (state machine, state, event), and (state machine, state, event, time). The first type of tuple queries for the occurrence of a specific state in a particular state machine during any time (i.e., no time is specified). The second type is the same as the first except that it also specifies a time. The time can be either an instant or a time interval. The third type of tuple queries for the occurrence of a specific event in a specific state in a particular state machine. The fourth type is similar to the third one, but it also specifies a time. When using the third and fourth types of tuples involving events, the associated time must be a time interval. The outcome of a predicate at a particular time is called a *predicate value*. The predicate applied to the global timeline generated in the analysis phase is called a *predicate value timeline*. As explained in Chapter 2, each event has two time bounds on the global timeline. To compute the predicate value timeline, the predicate is evaluated at each of these time bounds. The predicate value

Global Timeline			
State Machine	Begin State	Event	Time
StateMachine5	State5	Event5	11.2
StateMachine1	State0	Event1	12.4
StateMachine6	State5	Event6	13.1
StateMachine1	State1	Event2	18.9
StateMachine6	State6	Event7	20
StateMachine5	State5	Event5	21.4
StateMachine3	State3	Event3	22.3
StateMachine3	State4	Event4	26.3
StateMachine2	State0	Event8	30.9
StateMachine5	State5	Event5	31.2
StateMachine2	State2	Event9	32.3
StateMachine6	State4	Event10	32.3
StateMachine2	State1	Event12	35.6
StateMachine6	State6	Event11	37.9
StateMachine2	State2	Event13	38.9
StateMachine5	State5	Event5	40.6

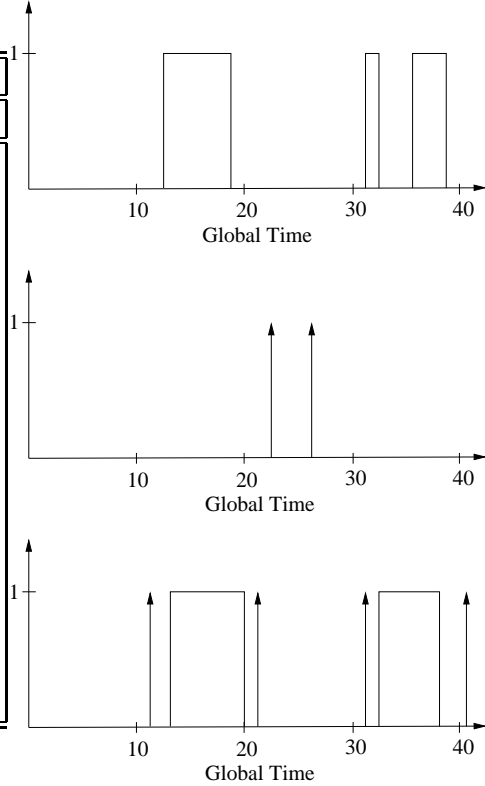


Figure 4.2: Predicate Value Timeline Example

timeline thus obtained contains a combination of impulses and steps.

Examples of predicates are given by the following expressions:

1.  $((\text{StateMachine1}, \text{State1}, 10 < t < 20) \mid (\text{StateMachine2}, \text{State2}, 30 < t < 40))$ : the predicate is true during any time between 10 and 20 ms when StateMachine1 is in State1, and during any time between 30 and 40 ms when StateMachine2 is in State2. During all other times, the predicate is false.
2.  $((\text{StateMachine3}, \text{State3}, \text{Event3}, 10 < t < 30) \mid (\text{StateMachine3}, \text{State4}, \text{Event4}, 20 < t < 40))$ : the predicate is true at any time between 10 and 30 ms when StateMachine3 is in State3 and Event3 occurs, and at any time between 20 and 40 ms when StateMachine3 is in State4 and Event4 occurs.
3.  $((\text{StateMachine5}, \text{State5}, \text{Event5}) \mid (\text{StateMachine6}, \text{State6}, 10 < t < 40))$ : the predicate is true whenever StateMachine5 is in State5 and Event5 occurs, and dur-

ing any time between 10 and 40 ms when StateMachine6 is in State6.

Figure 4.2 gives an example of a global timeline and shows the predicate value timelines obtained on applying the above predicates to the global timeline. (The time bounds for each event in the above global timeline are very close to each other. Therefore, in the above figure, the predicate is evaluated only at the mean of the two time bounds.)

### 4.3.2 Observation Function

For each defined predicate, the user must specify an observation function. The observation function is used by the user to extract the required information from the predicate value timeline as a single value. The input to an observation function is a predicate value timeline. The output of the function is called an *observation function value*. There are two types of observation functions, namely predefined functions and user-defined functions. The predefined observation functions in Loki are as follows:

1. `count(<U, D, B>, <I, S, B>, START, END)` : returns the number of times a transition from false to true (“U” up-transition), a transition from true to false (“D” down-transition), or both (“B”) occur, considering only impulses (“I”), only steps (“S”), or both (“B”) during the interval [START, END].
2. `outcome(t)` : returns the outcome of the predicate value at instant  $t$  (the outcome will be an integer, either 0 associated with false or 1 associated with true).
3. `duration(<T, F>, x, START, END)` : returns the time duration during the interval [START, END] for which the predicate is true (“T”) after the  $x$ th time a transition from false to true occurred, or the time duration for which the predicate is false (“F”) after the  $x$ th time a transition from true to false occurred.
4. `instant(<U, D, B>, <I, S, B>, x, START, END)` : returns the instant corresponding to the  $x$ th time a transition from false to true or a transition from true to false or

both occurred, considering impulses, steps, or both during the interval [START, END].

5. `total_duration(<T, F>, START, END)` : returns the total length of time during which the predicate value is true or false during the interval [START, END].

If a user wants to specify a measure that cannot be specified using one of the predefined functions, he/she can define his/her own observation function. In our implementation, a user-defined observation function is any function that can combine predefined observation functions with standard mathematical functions and can be compiled with a standard C compiler.

The following expressions are three examples of observation functions defined using predefined observation functions. The result obtained on applying them to the three predicate value timelines of Figure 4.2 is also given for each example.

- `count(U, B, 10, 35)` counts the number of times an up transition occurs between 10 and 35 ms considering both steps and impulses. The results obtained for the three predicate value timelines are, respectively, 2, 2, and 5.
- `duration(T, 2, 10, 40)` returns the length of time during which the predicate is true after the second transition from false to true occurred. For the three predicate value timelines, the obtained values are 1.4 ms, 0 ms, and 7.0 ms respectively.
- `instant(U, I, 2, 0, 50)` indicates the instant of the second up transition between 0 and 50 ms (considering only impulses). The results for the three predicate value timelines of Figure 4.2 are, respectively, 0 ms, 26.3 ms, and 21.2 ms.

### 4.3.3 Subset Selection

As explained above, a predicate and an observation function are defined for all experiments included in the study. After obtaining the predicate value timelines and the associated observation function values, a user might be interested in estimating a measure from a

subset of experiments of the study. Loki provides the user with the ability to select a subset of experiments based on the observation function values. Using standard mathematical functions that can be compiled with a standard C compiler and observation function values, the user can define a subset function that returns true or false. Examples of subset selections are:

- experiments for which the observation function value is between 2 and 10, and
- experiments that have positive observation function values.

#### 4.3.4 Study-Level Measures

The three concepts used for measures at the study level, namely predicates, observation functions, and subset selections, have been described above. This section explains the process of combining these three concepts to obtain measures at the study level.

Suppose the user defined a triple (subset selection, predicate, observation function). After defining this triple, the user might want to focus on a subset of experiments that is based on the observation function values obtained upon applying this triple to all the experiments. For this subset, a new predicate and a new observation function can be defined. This can be represented as the following sequence of triples: ((subset selection1, predicate1, observation function1), (subset selection2, predicate2, observation function2)). This process of selecting a subset and defining a new predicate and new observation function can be repeated. A measure at the study level is then defined by an ordered sequence of (subset selection, predicate, observation function) triples, where the subset selection of the first triple selects all the experiments in the study. The output of the last observation function of the ordered sequence is called a *final observation function value*.

## 4.4 Measures Defined Across Studies

Final observation function values are processed inside each study and across studies to obtain the campaign measure estimation. The campaign measure could be completely characterized if its probability distribution could be obtained. However, in practice, the distribution cannot be calculated. Therefore, for all practical purposes, knowledge of the moments is equivalent to knowledge of the distribution function, in the sense that it should *theoretically* be possible to exhibit all the properties of the distribution in terms of the moments [12] (pp. 108-109). In practice, the properties obtained when calculating the first four moments are very close to the properties of the real distribution.

The user can define campaign measures of three types: simple sampling, stratified weighted, or stratified user. We now focus on each measure type and the statistical estimations associated with it.

### 4.4.1 Simple Sampling Measures

Simple sampling measures in Loki are used when the user does not want to differentiate between the final observation function values of different studies. These measures are obtained by considering all the selected studies to be similar such that the final observation function values (associated with all experiments of all the selected studies) are contained in a single sample, i.e., they are all instances of the same random variable. In the following discussion, let  $M$  be the number of studies,  $n_i$  be the number of experiments in study  $i$ ,  $N = \sum_{i=1}^M n_i$  be the total number of experiments in all the studies, and  $x_{j,i}$  be the final observation function value of the  $j$ th experiment of study  $i$ .

The first four non-central moments are then defined by the following expressions:

$$\mu'_k = \frac{1}{N} \sum_{i=1}^M \sum_{j=1}^{n_i} x_{j,i}^k \quad \text{where } k = 1, 2, 3, 4$$

The three central moments of orders 2, 3, and 4 can be obtained from the first four non-central moments by the expressions in [13] p. 18, Eqn. (100):

$$\mu_2 = \mu'_2 - (\mu'_1)^2 \quad (4.1)$$

$$\mu_3 = \mu'_3 - 3\mu'_2\mu'_1 + 2(\mu'_1)^3 \quad (4.2)$$

$$\mu_4 = \mu'_4 - 4\mu'_3\mu'_1 + 6\mu'_2(\mu'_1)^2 - 3(\mu'_1)^4 \quad (4.3)$$

Once the first four central moments are obtained, the *skewness* and *kurtosis* coefficients are calculated using the following expressions:

$$\beta_1 = \frac{(\mu_3)^2}{(\mu_2)^3} \quad \beta_2 = \frac{\mu_4}{(\mu_2)^2} \quad (4.4)$$

Finally, percentiles for various  $\alpha$ -levels are obtained by using the Bowman and Shenton approximation [14, 15]. Bowman and Shenton introduced a rational fraction approximation for any percentile  $y_\gamma$  of a standardized distribution ( $\mu_1 = 0, \mu_2 = 1$ ) of the Pearson system. This approximation uses a 19-point formula:

$$y_\gamma = \frac{\pi_{\gamma,1}(\sqrt{\beta_1}, \beta_2)}{\pi_{\gamma,2}(\sqrt{\beta_1}, \beta_2)} \quad \text{where,} \quad (4.5)$$

$$\pi_{\gamma,i}(\sqrt{\beta_1}, \beta_2) = \sum_{0 \leq r+s \leq 3} \sum a_{\gamma,r,s}^{(i)} (\sqrt{\beta_1})^r \beta_2^s \quad \text{for } i = 1, 2 \quad (4.6)$$

The values of  $a_{\gamma,r,s}^{(i)}$  are given in [14, 15]. When  $\mu_3 \geq 0$ , the  $\gamma$ -percentile of the non-standardized distribution is given by  $z_\gamma = \mu_1 + \sqrt{\mu_2} y_\gamma$ . When  $\mu_3 < 0$ ,  $z_\gamma = \mu_1 - \sqrt{\mu_2} y_{(1-\gamma)}$ .

#### 4.4.2 Stratified Weighted Measures

Stratified weighted campaign measures are estimated by building samples containing the final observation function values for each selected study, computing the moments, and then using a weighted function to combine the estimations of moments obtained for each study. There

are several practical and statistical reasons for focusing on stratified weighted measures when evaluating fault tolerance mechanisms. For example, one very important parameter is the coverage of a fault tolerance mechanism [16]. When considering several independent studies, the overall coverage of the fault tolerance mechanism is defined by a weighted function across studies [17]. From a statistical viewpoint, the functions for calculating the moments are linear for linearly weighted combinations of the random variables representing the final observation function values, i.e.,  $f(ax + by) = cf(x) + df(y)$ , where  $f$  is the function for calculating a moment,  $x$  and  $y$  are random variables representing final observation function values, and  $a, b, c$ , and  $d$  are constants. Here we assume that the powers of random variables representing the final observation function values are independent across studies. Another statistical reason for considering stratified weighted measures is that the mean, a fundamental statistical estimator, is a special case of them.

In addition to the notations introduced in Section 4.4.1, we define  $p_i$  as the normalized weight associated with the study  $i$ . Adapting the previous expressions to focus on each study, we define the first four non-central moments for study  $i$  by:

$$\mu'_{k,i} = \frac{1}{n_i} \sum_{j=1}^{n_i} x_{j,i}^k \quad \text{for } k = 1, 2, 3, 4$$

The central moments of orders 2, 3, and 4 are obtained for study  $i$  with equations similar to the ones used for the simple sampling measures (Eqns. (4.1), (4.2), and (4.3)):

$$\begin{aligned} \mu_{2,i} &= \mu'_{2,i} - (\mu'_{1,i})^2 \\ \mu_{3,i} &= \mu'_{3,i} - 3\mu'_{2,i}\mu'_{1,i} + 2(\mu'_{1,i})^3 \\ \mu_{4,i} &= \mu'_{4,i} - 4\mu'_{3,i}\mu'_{1,i} + 6\mu'_{2,i}(\mu'_{1,i})^2 - 3(\mu'_{1,i})^4 \end{aligned}$$

The central moments associated with the overall campaign measure are obtained by making the assumption that the random variables (and their powers) associated with the

various studies are independent from one another. The mean is obtained by  $\mu'_1 = \sum_{i=1}^M p_i \mu'_{1,i}$ .

Moreover, the central moments of orders 2, 3, and 4 are then given by the following expression:

$$\mu_k = \sum_{i=1}^M p_i^k \mu_{k,i} \quad \text{for } k = 2, 3, 4$$

Finally, the skewness and kurtosis coefficients and the percentile points are calculated as explained for the simple sampling measures in Eqns. (4.4) and (4.6).

### 4.4.3 Stratified User Measures

A stratified user measure is a stratified measure in which the final observation function values of different studies are combined using a user-defined function other than a linearly weighted function. In case the user would like to define a combined campaign measure other than a linearly weighted function, the statistical features presented above can no longer be used. Indeed, the calculation of the first four moments associated with the campaign measure is not a trivial task for any arbitrary function that combines final observation function values of the various studies. The only result Loki gives for stratified user measures is a campaign measure in which each final observation function value in the user-defined function is replaced by the mean of the final observation function values for that study. However, the campaign measure value thus obtained may have no statistical meaning.

# Chapter 5

## Example of a Fault Injection Campaign

### 5.1 Introduction

The main goals of the Loki fault injection tool are fault removal and the evaluation of the dependability and performance of distributed systems (fault forecasting). The success of Loki in achieving these goals largely depends on how well it performs the required fault injections in the system under study, while at the same time being very easy to use. This chapter is intended to provide a step-by-step explanation of how a user can use Loki to inject faults into his/her system and obtain the required measures. For that purpose, a test application that resembles many real-life applications is selected as an example, and each step in the process of fault-injecting it using Loki is described in detail.

### 5.2 Test Application

Quite obviously, the first step in fault-injecting any application is to have an implementation of the application itself. The test application chosen for illustration of some of the capabilities of Loki is a simple leader election application. Leader election is an important component of many real-life distributed systems, such as group communication systems.

The application consists of  $n$  processes, and the function of the application is that the

processes elect a leader from amongst themselves. To do this, each process picks a random number and sends it to the remaining  $n - 1$  processes. At the end of that round, each process has  $n$  numbers, and it selects the process that picked the highest number as the leader. In case of ties, this arbitration is repeated until it is resolved. When the current leader fails by crashing, the remaining processes elect a new leader using the same protocol. Crashed processes can restart and join the system again.

### 5.3 State Machine Specification

After implementing the application, the next step is to specify the execution of the application as a state machine at the desired level of abstraction required for fault injection. In the case of the test application, the components of the application (i.e., all of the  $n$  processes) have identical state machine abstractions. Note that in other applications, different components of the distributed application might have different state machine abstractions. The state machine abstraction of the test application components is shown as a graph in Figure 5.1.

In the state machine abstraction of Figure 5.1, each vertex is labeled with the state name, and each arc is labeled with the local event notification corresponding to it.

At the start of the application, the state machine corresponding to each process is in the BEGIN state. If the process is a new one, a START event is generated within it, and its state machine transitions to the INIT state. If it is a restarted one, a RESTART event is generated within it, and its state machine transitions to the RESTART\_SM state. When a RESTART\_DONE event is generated in the RESTART\_SM state, the state machine transitions to the FOLLOW state, since a restarted process will always be a follower. The INIT state represents the initialization of the processes, which includes the setting up of communication between the processes. After the initialization, an INIT\_DONE local event notification is generated in the process, and its state machine transitions to the ELECT

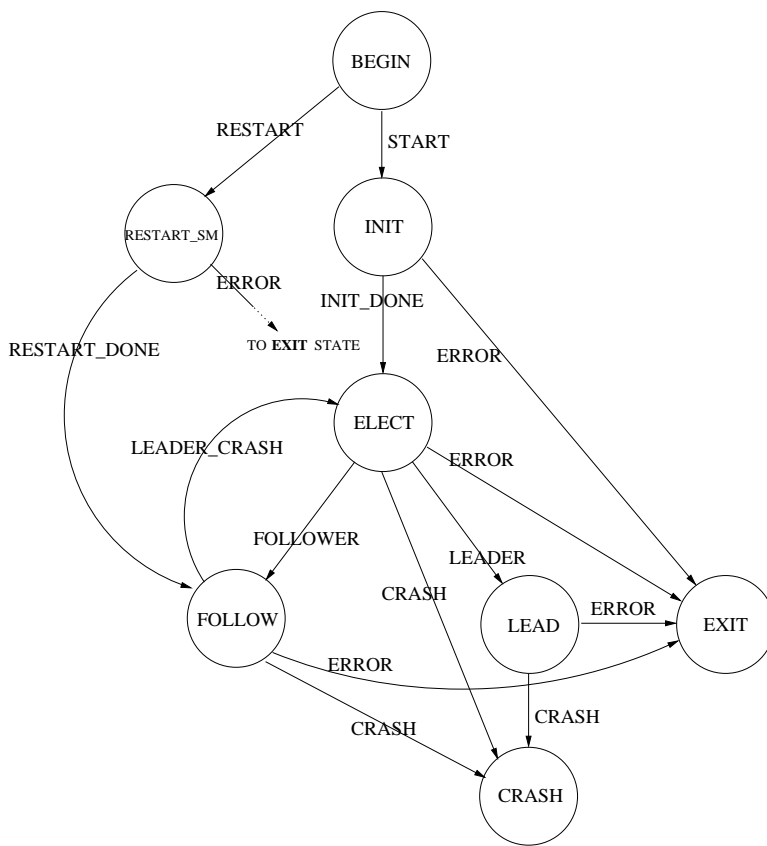


Figure 5.1: Election Protocol

state. The ELECT state signifies that the processes are performing the leader election. At the end of the election, the state machine associated with the leader receives a LEADER notification and moves to the LEAD state, while all the other state machines receive a FOLLOWER notification and move to the FOLLOW state. If an error occurs in any of the states other than the BEGIN and CRASH states, an ERROR notification is received by the state machine, which then transitions to the EXIT state. When the leader crashes, a LEADER\_CRASH notification is received by the state machines of all the follower processes. The state machine of the leader process receives a CRASH event notification and transitions to the CRASH state. All the other state machines transition to the ELECT state, signifying the start of a new leader election. If a CRASH event is received in the FOLLOW or ELECT states, the state machine transitions to the CRASH state.

The state machine abstraction given in Figure 5.1 has to be converted into a script-like textual format before it can be given as input to the Loki runtime. However, before that can be done, the number of components in the application has to be decided upon, and their corresponding state machines should be given unique names. These nicknames are used in referring to the state machines subsequently. For the test application, assume that there are three components (processes) and that their corresponding state machines are named black, yellow, and green. Even though the state machine abstraction is the same for black, yellow, and green, their textual state machine specifications might be different. This is because the list of state machines to be notified on a state change might be different for each of them. The notify list of the state machines is used to maintain the partial view of global state necessary for fault injection, and hence depends on the fault specification (as described in Section 5.4). The fault specification specifies the set of global states in which each fault should be injected. The notify lists of state machines are obtained by observing the fault specifications of all the components. This process of obtaining the notify lists could possibly be automated in future versions of Loki.

The state machine specification for the state machine black is given below:

global\_state\_list

BEGIN

INIT

RESTART\_SM

ELECT

FOLLOW

LEAD

CRASH

EXIT

end\_global\_state\_list

event\_list

START

INIT\_DONE

RESTART

RESTART\_DONE

LEADER

FOLLOWER

LEADER\_CRASH

CRASH

ERROR

end\_event\_list

state INIT notify green yellow

INIT\_DONE ELECT

ERROR EXIT

```
state RESTART_SM notify green yellow
```

```
RESTART_DONE FOLLOW
```

```
ERROR EXIT
```

```
state ELECT notify
```

```
FOLLOWER FOLLOW
```

```
LEADER LEAD
```

```
CRASH CRASH
```

```
ERROR EXIT
```

```
state LEAD notify
```

```
CRASH CRASH
```

```
ERROR EXIT
```

```
state FOLLOW notify
```

```
LEADER_CRASH ELECT
```

```
CRASH CRASH
```

```
ERROR EXIT
```

```
state CRASH notify green yellow
```

```
state EXIT notify
```

The state machine specifications for green and yellow are similar to that of black except that the notify lists are different. The differences for green are as follows:

```
state INIT notify black yellow
state RESTART_SM notify black yellow
state CRASH notify black yellow
```

The differences for yellow are as follows:

```
state INIT notify black green
state RESTART_SM notify black green
state CRASH notify black green
```

## 5.4 Fault Specification

The fault specification stipulates the global state in which each fault should be injected. The fault specification depends on the kind of faults to be injected into the application. The actual fault injection code is implemented by the user in the probe during application instrumentation. The Loki runtime itself does not need to know the kind and type of each fault or the method by which it is injected by the probe. The type of faults to be injected depends on the kind of evaluation the user intends to perform on the application.

For the test application, two example evaluations are as follows. The first evaluation involves determining the coverage of an error in the leader. This is done by the injection of a fault into the leader process; if this fault becomes an error and crashes the leader, the probability that the leader process recovers is determined. The following fault expressions are defined for this evaluation:

```
bfault1 (black:LEAD) always in state machine black
gfault1 (green:LEAD) always in state machine green
yfault1 (yellow:LEAD) always in state machine yellow
```

The second evaluation involves finding the correlation between an error in the leader (leading to a crash), and a simultaneous error in another process. This is done by injecting a fault into one of the follower processes when the leader crashes and then observing the effect. The fraction of time the injected fault becomes an error is compared with the fraction of time an injected fault becomes an error if it is not simultaneous with an error in the leader. If leader is black and the follower is green, we have the following fault specification in addition to `bfault1`. Fault specifications for the other cases can be specified similarly.

```

gfault2    ((black:CRASH) & ((green:FOLLOW) | (green:ELECT)))    once in state ma-
chine green
gfault3    ((green:FOLLOW) | (green:ELECT))    once in state machine green

```

Note that even though the state machine green changes state from FOLLOW to ELECT when state machine black crashes as a leader, the fault `gfault2` is injected only once. This is because the fault parser in Loki is positive-edge-triggered, i.e., it injects a fault only when the value of the Boolean fault expression transitions from a false to a true. Also, note that the type of fault injected is completely left to the user; for example, `bfault1` could be a corruption of a random location in the process's stack. Furthermore, the state machine specification given in Section 5.3 takes into consideration all of the above fault expressions.

## 5.5 Instrumentation and Probe Design

Since the test application involves the crashing and restarting of processes (state machines), the original Loki runtime cannot be used to evaluate it. The new runtime has to be used for this evaluation. Since the source code of the application is available, the probe code is made a part of the application source code. First the `main()` function of the application is renamed

as `appMain()`. The `notifyEvent()` method of the state machine is used by the probe code to send local event notifications to the state machine. The first call to `notifyEvent()` is used to set the initial state of the new or restarted state machine. Subsequent calls, at appropriate locations where the local state transitions occur in the code, are used to send local event notifications corresponding to the local state transitions. After being instrumented in this fashion, the application is compiled with the Loki library. The Loki library contains the code for the state machine, state machine transport, fault parser, and recorder. The instrumented application code is shown below:

```
#include 'Probe.h'

...

void appMain(int argv, char * argv[]){

...

    if(new){ /* New state machine */

        notifyEvent(INIT); /* Initialize state of state machine */

        /* Perform application initialization */

        notifyEvent(INIT_DONE);

        /* Perform election */

        if(leader)

            notifyEvent(LEADER);

        else

            notifyEvent(FOLLOWER);

    } else { /* Restarted state machine */

        notifyEvent(RESTART); /* Initialize state of state machine */

        /* Perform any application initialization on restart */

        notifyEvent(RESTART_DONE);

    }

}
```

```

do{
    /* Wait until leader crashes */
    notifyEvent(LEADER_CRASH);
    /* Perform election */
    if (leader)
        notifyEvent(LEADER);
    else
        notifyEvent(FOLLOWER);
} while(true);
}

```

```

void sigHandler(int signal){
    /* Perform cleanup */
    notifyEvent(CRASH);
    notifyOnCrash();
    exit(-1);
}

```

## 5.6 Campaign Execution

Before the instrumented application can be used in experiment runs for a study, a few files have to be specified. First, a list of host names being used in the campaign execution should be specified, one per line, in a machines file. Then, for each state machine (in a study), a study file has to be specified, which has the following format:

```

<SMNickName>
<NodeFile>

```

```
<StateMachineSpecificationFile>  
<FaultSpecificationFile>  
<InstrumentedApplicationExecutable Path>  
<ApplicationArguments>
```

Now the campaign execution can begin. First, the central daemon is started, and then the local daemons are started. The command to start the local daemons is

```
lokid <DaemonStartUpFile> <DaemonContactFile> <NodeFile> <CentralDaemonHostName>  
<CentralDaemonPort>
```

Then all the state machines are started. The command to start each of them is

```
InstrumentedApplicationExecutable <StudyFile> <DaemonContactFile> <LocalTimelineFile>
```

The experiment is allowed to run until completion. The output of each experiment execution consists of the local timelines of the state machines. Before and after every experiment, timestamps are obtained, which are used for off-line clock synchronization during the analysis phase. All the timestamps are stored together in a single timestamps file. To obtain the timestamps, the following command is run on all the machines in the system:

```
getstamps <MachinesFile> <NumberOfSyncMsgs> <TimeBetweenSyncMsgs> <PortNumber>  
<TimestampsFile>
```

If multiple studies are present in a fault injection campaign, the execution procedure for each of them is the same as above. In the case of the test application, multiple studies are executed to perform the two evaluations suggested in Section 5.4. More details about these

studies are given in Section 5.8.

## 5.7 Campaign Analysis

In campaign analysis, all the local timelines generated during the campaign execution are converted to the corresponding global timelines. The first step in doing this is to compute the bounds on  $\alpha$  and  $\beta$  for each machine. This is done using the following command:

```
alphabet <TimestampsFile> <MachinesFile> <AlphabetFile> <MHzFile>
```

The  $\alpha$  and  $\beta$  of each machine are computed using the fastest machine in the system as the reference machine, and are stored in the alphabet file. The fastest machine is taken as the reference machine, since there would be a loss of accuracy if the times on a fast machine were mapped onto the times of a slower machine. The MHz file contains the speed of the fastest machine in the system. After computing the  $\alpha$  and  $\beta$  for each machine, the local timelines of all the state machines for each experiment are placed into a single global timeline, so that there is one global timeline per experiment. Also, the correctness of the fault injections is determined. The following command performs both these operations.

```
makeglobal <AlphabetFile> <MHzFile> <GlobalTimelineFile> <LocalTimelineFile 1>  
<FaultInjectionResultsFile 1> ... <LocalTimelineFile n> <FaultInjectionResultsFile  
n>
```

The  $i$ th local timeline file is the local timeline of the  $i$ th state machine for this experiment. The fault injection results files contain indications of whether each fault has been correctly injected in the corresponding state machine in this experiment. If any of the faults are incorrect in an experiment, the experiment is discarded, and is not used in measure com-

putation. Note that the above campaign analysis is performed for each experiment within each study. For the test application, the campaign execution and analysis are performed as described above, and the global timelines are obtained for each of the experiments in which fault injections were done correctly.

## 5.8 Measure Specification and Estimation

This section describes the method of obtaining the required measures from the global timelines using Loki. For the test application, the first measure corresponds to the first intended evaluation given in Section 5.4, i.e., determining the coverage of an error in a leader process. Let studies 1, 2, and 3 have only faults `bfault1`, `gfault1`, and `yfault1` injected, respectively. Consider study 1, in which `bfault1` is injected into the state machine `black` when it becomes a leader. To determine the coverage of an error caused by `bfault1`<sup>1</sup>, the following study measure can be used:

```
((default, (black:CRASH), total_duration(T, START_EXP, END_EXP)), ((OBS_VALUE > 0), (black:RESTART_SM), (total_duration(T, START_EXP, END_EXP) > 0)))
```

In the above study measure, `START_EXP` and `END_EXP` are Loki macros that take the values of the beginning time and ending time of the current experiment, respectively. The macro `OBS_VALUE` is the observation function value of the observation function of the previous triple. In the first triple, the `default` subset selection selects all the experiments, and the predicate and observation function determine the time spent by the state machine `black` in the `CRASH` state as the observation function value. The subset selection of the second triple checks whether this observation function value is positive, to filter out all the experiments in which `bfault1` has not caused an error and a subsequent crash in the state machine `black`.

---

<sup>1</sup>It is assumed that the effect of the error is to crash the process and that it does not cause the process to behave maliciously.

The predicate and the observation function of the second triple check whether the crashed black state machine has been restarted (thus covering the crash failure). Thus, the above study measure returns a 1 for an experiment if an error has resulted from the fault injection and has been covered, and it returns a 0 for an experiment if the error has occurred but has not been covered. Similar study measures, involving fault injections into the green and yellow state machines, can be defined for studies 2 and 3.

Assume that the typical fault occurrence rates,  $w_b$ ,  $w_g$ , and  $w_y$ , are known for the state machines black, green, and yellow, respectively. Also, assume that the coverages for black, green, and yellow are  $c_b$ ,  $c_g$ , and  $c_y$ , respectively. Then the overall coverage of the system,  $c$ , is as follows:

$$c = \frac{w_b c_b + w_g c_g + w_y c_y}{w_b + w_g + w_y}$$

Thus, given the fault occurrence rates, the overall coverage of the system can be estimated from the study measures of studies 1, 2, and 3, using the statistical methods for stratified weighted measures as described in Chapter 4. Note that here we assume that the faults follow a “representative sample.” This means that we assume that during the fault injection process, the faults with a higher fault occurrence rate are injected in a greater number than those with a lower fault occurrence rate.

The second measure corresponds to the second evaluation, and estimates the correlation between a leader crash and a simultaneous error in another follower process. Consider study 4, in which `bfault1` and `gfault2` are injected. The study measure can be defined as follows:

```
((default, (black:CRASH), total_duration(T, START_EXP, END_EXP)), ((OBS_VALUE > 0), (green:CRASH), (total_duration(T, START_EXP, END_EXP) > 0)))
```

As in the previous study measures, the first triple gives the time spent by the black

state machine in the CRASH state. The subset selection of the second triple filters out all the experiments in which `bfault1` has not caused a crash in the black state machine. The predicate and observation function check whether the fault `gfault2` has caused the green state machine to crash. Thus, this study measure returns a 1 if the black state machine has crashed and `gfault2` crashed the green state machine, and it returns a 0 if the black state machine crashed and the green state machine did not crash. Computing the average of this study measure gives the fraction of injections of `gfault2` that caused errors (i.e., crashes), given that the leader process has already crashed. As a comparison, the fraction of faults in the green state machine that transform to errors when the leader process has not crashed can be computed by constructing a study 5 in which only the fault `gfault2` is injected, and then using the following study measure in a manner similar to the study measures described above:

```
(default, (green:CRASH), (total_duration(T, START_EXP, END_EXP) > 0))
```

This comparison gives an idea of the correlation of the leader crash with errors in other processes. Studies similar to studies 4 and 5 can be designed for other pairs of state machines, and study measures similar to the above can be obtained. These study measures can then be combined using the statistical techniques given in Chapter 4.

# Chapter 6

## Conclusions

This thesis presents Loki, a new tool for the evaluation of the dependability and performance of distributed systems. Loki uses fault injection to perform this evaluation. It facilitates fault injection based on a partial view of the global state of the system, i.e., the state of multiple components of the system. It then performs a check to verify whether the faults were injected in the correct global states, and discards all the experiments with incorrect fault injections. Accurate measures are then computed from the results of the correct fault injections. The work presented in this thesis includes the development of Loki's runtime and measures framework.

The Loki runtime allows the user to define state machine descriptions of the system's execution. The user also defines the fault specifications that specify the global state in which faults have to be injected in the various components of the system. The runtime of each node uses the state machine specifications, along with the local event notifications of the probe and remote state notifications, to maintain the partial view of global state. Whenever the partial view matches the global state needed for the injection of a fault, the runtime instructs the probe to inject the fault. The runtime records all the state changes and fault injections, along with their occurrence times, in local timelines. Note that there is one local timeline per node. Synchronization messages are passed before and after each fault injection experiment; these messages are used during off-line analysis to convert all the local timelines into a single global timeline. The fault injections are checked for correctness,

and the experiments with incorrect injections are discarded.

The measures framework in Loki gives the user flexibility in specifying a wide range of dependability and performance measures. Measures are specified at the study and campaign levels. Each measure at the study level is composed of a sequence of subset selection, predicate, and observation function triples. The study-level measures can be combined to obtain simple sampling, stratified weighted, or stratified user campaign-level measures. The framework also includes the computations to be performed on the results of the correct fault injection experiments to obtain accurate measure estimations.

In the future, it is important that Loki be used to evaluate a few real-life distributed systems so that the effectiveness of Loki as a distributed system evaluation tool can be assessed. The execution of each real-life application will need to be specified as a set of state machines, and a set of fault injections and measures will have to be decided upon. Then the application has to be instrumented and experiments run to evaluate the system and, at the same time, assess the effectiveness of Loki in injecting faults in the right global states and in accurately obtaining the required measures. The process of instrumenting the application will afford another opportunity for possible future work, namely in developing probe templates for a variety of common fault types, such as memory, CPU, and communication faults.

Furthermore, in future work, a performance analysis of all the Loki runtime components could be conducted and the performance could be improved, depending on the efficiency of the runtime in injecting faults. For example, the fault parser algorithm could be optimized, and techniques could be developed to allow state machines to send state change hints to remote state machines well in advance of the actual state changes so as to increase the efficiency of fault injection. Similarly, the measures framework could be enhanced based on experience with real-life systems. Finally, the concepts developed in Loki could be applied to other areas, such as online debugging of distributed programs using their global states, and detection of security violations in distributed systems.

# References

- [1] J. Laprie, “Dependability of computer systems: Concepts, limits, improvements,” in *Proceedings of the 25<sup>th</sup> International Symposium on Fault-Tolerant Computing, Special Issue*, pp. 42–54, 1995.
- [2] F. Lange, R. Kroeger, and M. Gergeleit, “JEWEL: Design and implementation of a distributed measurement system,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 657–671, November 1992.
- [3] G. Alvarez and F. Cristian, “Centralized failure injection for distributed, fault-tolerant protocol testing,” in *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems (ICDCS’97)*, pp. 78–85, May 1997.
- [4] S. Han, K. G. Shin, and H. A. Rosenberg, “DOCTOR: An integrated software fault injection environment for distributed real-time systems,” in *Proceedings of the International Computer Performance and Dependability Symposium*, pp. 204–213, 1995.
- [5] K. Echtele and M. Leu, “The EFA fault injector for fault-tolerant distributed system testing,” in *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 28–35, 1992.
- [6] S. Dawson, F. Jahanian, T. Mitton, and T. L. Tung, “Testing of fault-tolerant and real-time distributed systems via protocol fault injection,” in *Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, pp. 404–414, June 1996.

- [7] D. Bhatt, R. Jha, T. Steeves, R. Bhatt, and D. Wills, “SPI: An instrumentation development environment for parallel/distributed systems,” in *Proceedings of the 9th International Parallel Processing Symposium*, pp. 494–501, 1995.
- [8] D. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. Iyer, “NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors,” in *Proceedings of the 4th IEEE International Computer Performance and Dependability Symposium (IPDS-2K)*, pp. 91–100, March 2000.
- [9] D. A. Henke, “Loki – an empirical evaluation tool for distributed systems: The experiment analysis framework,” Master’s thesis, University of Illinois at Urbana-Champaign, 1998.
- [10] M. Cukier, R. Chandra, D. Henke, J. Pistole, and W. H. Sanders, “Fault injection based on the partial global state of a distributed system,” in *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pp. 168–177, October 1999.
- [11] R. Chandra, M. Cukier, K. R. Joshi, R. M. Lefever, and W. H. Sanders, *Loki User’s Manual – version 1.0*. Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, August 2000.
- [12] A. Stuart and J. K. Ord, *Distribution Theory, Kendall’s Advanced Theory of Statistics, 1*. Edward Arnold, London, 1987.
- [13] N. L. Johnson and S. Kotz, *Distributions in Statistics – Continuous Univariate Distributions-1*. John Wiley & Sons, New York, 1969.
- [14] K. O. Bowman and L. R. Shenton, “Approximate percentage points for Pearson distributions,” *Biometrika*, vol. 66, no. 1, pp. 147–151, 1979.

- [15] K. O. Bowman and L. R. Shenton, “Further approximate Pearson percentage points and Cornish-Fisher,” *Communications in Statistics, Simulation, and Computation*, vol. B8, no. 3, pp. 231–244, 1979.
- [16] W. G. Bouricius, W. C. Carter, D. C. Jessep, P. R. Schneider, and A. B. Wadia, “Reliability modeling for fault-tolerant computers,” *IEEE Transactions on Computers*, vol. 20, no. 11, pp. 1306–1311, 1971.
- [17] D. Powell, E. Martins, J. Arlat, and Y. Crouzet, “Estimators for fault tolerance coverage evaluation,” in *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pp. 228–237, 1993.
- [18] R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders, “Loki: A state-driven fault injector for distributed systems,” in *Proceedings of the International Conference on Dependable Systems and Networks (FTCS-30)*, pp. 237–242, June 2000.
- [19] M. Cukier, D. Powell, and J. Arlat, “Coverage estimation methods for stratified fault-injection,” *IEEE Transactions on Computers*, vol. 48, pp. 707–723, July 1999.
- [20] J. L. Pistole, “Loki – an empirical evaluation tool for distributed systems: The run-time experiment framework,” Master’s thesis, University of Illinois at Urbana-Champaign, 1998.