From *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, Nürnberg, Germany, October 16-18, 2000, pp. 248-257.

Dynamic Node Management and Measure Estimation in a State-Driven Fault Injector*

Ramesh Chandra, Michel Cukier, Ryan M. Lefever, and William H. Sanders

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory and Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, Urbana, Illinois 61801, USA
{ramesh,cukier,lefever,whs}@crhc.uiuc.edu

Abstract

Validation of distributed systems using fault injection is difficult because of their inherent complexity, lack of a global clock, and lack of an easily accessible notion of a global state. To address these challenges, the Loki fault injector injects faults based on a partial view of the global state of a distributed system, and performs a post-runtime analysis using an off-line clock synchronization algorithm to determine whether the faults were properly injected. In this paper, we first describe an enhanced runtime architecture for the Loki fault injector and then present a new method for obtaining measures in Loki. The enhanced runtime allows dynamic entry and exit of nodes in the system. It also offers more efficient multicast of notification messages and more efficient communication between state machines on the same host, and is more scalable than the previous runtime. We then detail a new and flexible method for obtaining a wide range of performance and dependability measures in Loki.

Keywords : *Dependable distributed systems, Validation, Fault injection, Measure language, Statistical measure estimation.*

1. Introduction

Distributed systems are being increasingly used in building critical applications and hence validating their dependability is an important activity. Fault injection is an important and effective way to assess the dependability of such systems. Since faults occurring in a distributed system can depend on its global state, a fault injector needs to keep track of the system's global state to inject realistic faults, which makes its task difficult and challenging.

Several existing fault injection and measurement tools (including EFA [9], Orchestra [8], SPI [2], NFTAPE [13], DOCTOR [10], and CESIUM [1]) have focused on distributed systems. These tools work well for their intended purposes. However, they do not support injections based on the global state of a distributed system and do not provide means to obtain statistical measures from the fault injection experiments. With these issues in mind, we have developed a global state-driven fault injector called *Loki* [6, 7]. Loki can inject faults in a distributed system based on a partial view of its global state obtained using notifications, and can determine, using a post-runtime analysis, whether each fault was injected as intended. The results from the correct fault injections are then used to compute the measures specified by the user.

During fault injection in a distributed system, components of the system may crash and restart. However, the previous Loki runtime, as described in [7], did not support dynamic entry and exit of components of the system. To overcome this limitation of Loki, we redesigned the Loki runtime. The first part of the paper describes the design of the new Loki runtime. In particular, it presents a set of possible design choices and describes in detail the features of the new runtime implementation. The second part of the paper presents another new feature of Loki: statistical estimation of a wide array of measures. The measures desired from the fault injection results are typically closely related to the nature of the system under study; hence, the fault injector should give the user sufficient flexibility in specifying measures. Also, these measures must be statistically significant in order to be useful. Considering these issues, we have developed a new method for measure estimation in Loki that includes 1) a flexible measure language for specifying a wide range of performance and dependability mea-

*This research has been supported by DARPA Contract F30602-96-C-0315.

sures, and 2) the computations to be performed on the results from the correct fault injections to obtain statistical estimation for the specified measures. Together, these new functionalities significantly increase the capability of Loki.

The remainder of the paper is organized as follows. Section 2 introduces the basic concepts of Loki and gives a brief review of the Loki fault injector. Section 3 details the design of the enhanced Loki runtime. Section 4 presents the new method for measure estimation in Loki, including the flexible measure specification language, and the measure computation. Section 5 gives an example application that illustrates the use of the new runtime and measure estimation. Finally, Section 6 presents our conclusions and gives directions for future work.

2. Review of the Loki Fault Injector

In this section, we briefly review the Loki fault injector, which is described in [6]. We first present the concept of partial view of global state, which is central to Loki. We then introduce the concepts of a fault injection campaign, study, and experiment in Loki. Then we present the main phases in the process of evaluation of a distributed system using Loki.

The concept of *state* is fundamental to Loki. We assume that at the desired level of abstraction (for fault injection), the execution of a component of the distributed system under study can be specified as a state machine. 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. 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.

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 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. 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. Fault injection campaigns can be defined using Loki's graphical user interface, the *Loki interface* described in [6].

After the campaigns have been specified, the actual evaluation of the system has to be performed. Procedurally, the process of evaluating a system using Loki can be divided into three main phases, namely the runtime phase, the analysis phase, and the measure estimation phase. Note that each of these three phases is executed for each experiment within every study, in each campaign. These phases are briefly described below. For more details, refer to [7].

The runtime phase involves running the distributed system, injecting faults into the system at appropriate times, collecting observations, and recording them. In Loki, the distributed system (under study) is divided into basic components (i.e., processes) 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 runtime 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 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 runtime can be divided into two main parts: one that is independent of the system under study and one that is dependent on it. The *state machine*[1], *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 its functions that are mentioned above. For a detailed description of these units and their individual functions, refer to [7]. Note that the runtime only uses the necessary *state change notifications* between nodes to keep track of the partial view of the global state. Also, to be as non-intrusive to the system as possible, the runtime does not block the system while these notifications are in transit. Therefore, the system could change state while a notification is in transit: this implies that the partial view could sometimes be out-of-date. This could lead to incorrect fault injections and hence incorrect measures.

The analysis phase prevents such errors 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). Only the correct fault injec-

[1] Note that a state machine specification is a description that is dependent on the system under study, while the state machine is a component of the runtime that tracks the state given in the specification.

tions are used in computing the measures. The post-runtime check involves placing the local times from each of the nodes into a single global timeline, and then using the fault specifications to determine whether each fault was injected in the right state. Loki uses an *off-line clock synchronization* algorithm to translate the local times to a global timeline. This algorithm uses synchronization messages, which are generated by two synchronization-message-passing mini-phases before and after the runtime phase. These messages are non-intrusive, since they are generated when the application is not executing. The algorithm assumes that the drifts of the system clocks are linear. A more detailed explanation of the algorithm, along with its use in Loki, can be found in [7]. The measure estimation phase follows the analysis phase and involves computing statistically significant values for the user-specified measures. Section 4 of this paper presents in detail the measure estimation phase in Loki.

3. Enhanced Loki Runtime Architecture

In this section, we describe the new Loki runtime architecture. More specifically, we first consider the shortcomings of the previous Loki runtime and present the desirable features of the enhanced runtime. We then identify different designs we might have chosen for the enhanced architecture. Based on its advantages, we indicate which design we chose for implementation and then explain it in detail.

3.1. Shortcomings of the Previous Architecture

A detailed description of the original Loki runtime architecture can be found in [7]. The original version of the runtime provided the core functionality required for fault injection in Loki, as described in [7]. Our main goal in designing the previous version of the runtime was to prove the concepts underlying Loki. In addition, we wanted to get an efficient runtime implementation with very low intrusion and high efficiency in fault injection and measurement. We have empirically shown in [7] that using by this runtime, Loki was able to inject faults in the desired global state if the application stayed in the state for a time greater than a couple of OS time-slices.

However, a major shortcoming of the previous 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, during the fault injection of a general distributed system, processes can crash and restart, so the static nature of the previous runtime is an unacceptable limitation.

3.2. Design Choices for the Enhanced Architecture

As mentioned above, it was necessary for the enhanced runtime to support dynamic entry and exit of nodes while maintaining the low intrusion and high efficiency of the previous runtime. To do this, the transport mechanism of the Loki runtime needed to be redesigned so that a dynamically-entering node's state machine can establish communication with all the existing state machines with minimal intrusion to the system.

3.2.1. The Design Choices. We identified three possible high-level designs, shown in Figure 1, for the enhanced Loki runtime architecture. 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 and organization of daemons in the system and 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: the 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 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 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.

3.2.2. Comparison of the Design Choices. 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. However, the fully distributed design has a static list of nodes and hence only supports restarts on the same host: this would be very restrictive. In the centralized design, there could be a delay before

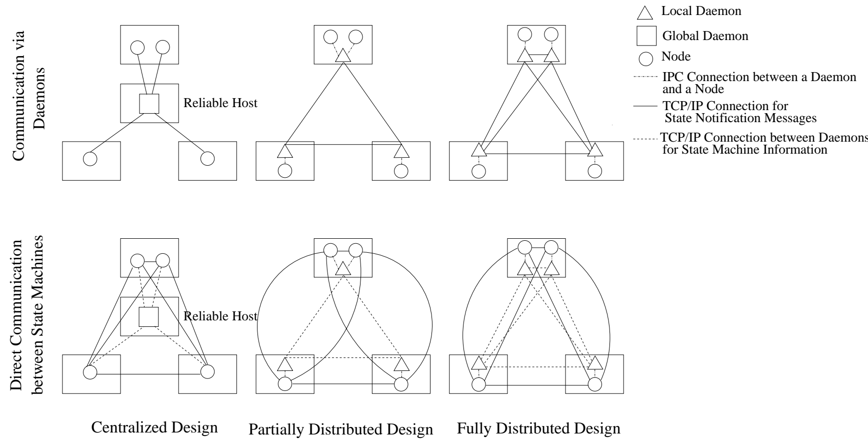

Figure 1. Design Choices for the Loki Runtime Architecture

the global daemon detects a node crash (because detection is by breaking of TCP/IP links). This would cause an incorrect time to be recorded for the crash event in the local timeline, and could lead to incorrect analysis and hence incorrect measure estimations. Additionally, the centralized design limits scalability. Hence, both the fully distributed design and the centralized design are not suitable for the enhanced Loki runtime.

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 better than direct communication between state machines. Indeed, 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. Also, in communication using daemons, if the runtime portion of a node becomes corrupted during fault injection, proper checks implemented at the local daemon could help contain the effect of the fault. Furthermore, since in current systems the IPC delay is of the order of 20μs and the TCP/IP delay is of the order of 150μs, 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.

3.3. The New Loki Runtime

In this section, we describe the new Loki runtime in detail. Specifically, we elaborate on the runtime architecture and describe how it works when a node starts and during the normal execution of a node. Then we describe what happens on a node crash and restart, and on a host reboot.

3.3.1. Architecture. Figure 2 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, we also have a central daemon to which all the local daemons are connected. This central daemon executes on a host not involved in fault injection, and is used to take care of host crashes and reboots during fault injection. Each node 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. The state machine keeps track of the partial view of the global state necessary for its node. It receives local state change notifications from the probe, and state change notifications of remote nodes from remote state machines.

The state machines of different nodes send state change notifications to each other using the state machine transport. The state machine transport is connected to the local daemon using shared memory. Signaling between the state machine and the local daemon, to indicate that the shared memory contains a message, is done using semaphores. The local daemons on different hosts are connected to each other and to the central daemon using TCP/IP. The state machine transports communicate the state change notifications via the local daemons.

The recorder records the state changes and fault injections along with their times of occurrence. In Loki, Boolean fault expressions are used to trigger fault injections. The fault parser parses these fault expressions on every state change and instructs the probe to inject the corresponding fault when an expression is satisfied. The probe in Loki is

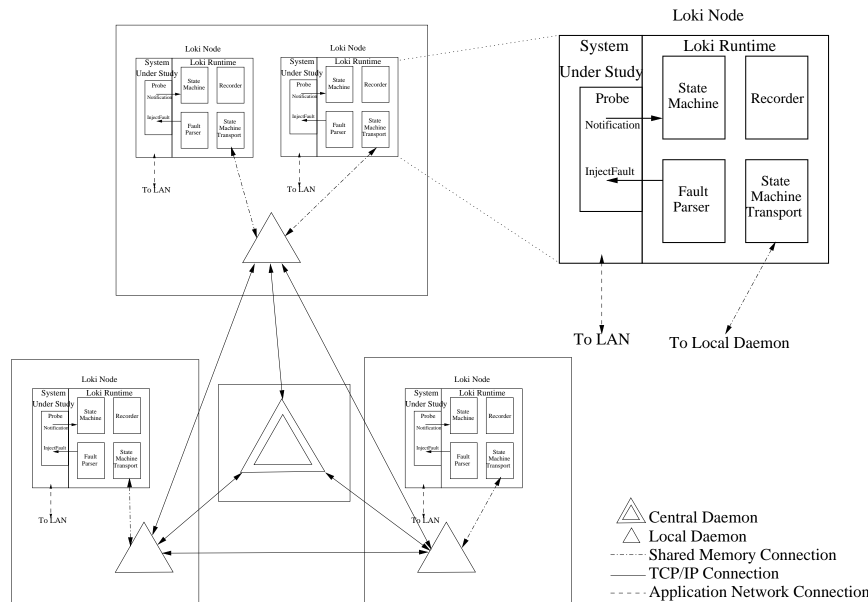

Figure 2. The New Loki Runtime Architecture

system-dependent and must be implemented by the system designer. It monitors the local node for state transitions and notifies the state machine of them. Also, it is the probe that performs the actual fault injections when the fault parser instructs it to do so.

3.3.2. 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 and waits for connection requests from the local state machine transports. The identifiers of these known shared memory regions are stored in a daemon-file. The central daemon then starts up only the nodes specified by the user in a startup file. 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-file, and sends a connection request to it. The daemon, on servicing the request, creates a new shared memory region along with the associated semaphores 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 forwards the notifications to them. These daemons in turn forward the notification to the 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 all the nodes of the system reach exit states, the experiment ends.

3.3.3. 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 due to an injected fault, the signal handler for the node deletes the shared memory region used to communicate with the local daemon and the associated semaphore. Because of this deletion, the local daemon is notified of the crash by the OS. The local daemon then writes the crash event to the local timeline of the crashed node's state machine and notifies all the other daemons of the crash.

3.3.4. 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 also contains the name of the host on which the state machine was restarted, which is used during off-line clock synchronization. Then it 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.

3.3.5. 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. They wait for the host to boot back up and reconnect to its local daemon. This support for host crash and reboot has not yet been implemented in Loki.

Global Timeline			
State Machine	Begin State	Event	Time
StateMachine3	State3	Event3	11.2
StateMachine1	State0	Event1	12.4
StateMachine1	State1	Event2	18.9
StateMachine3	State3	Event3	21.4
StateMachine2	State2	Event2	22.3
StateMachine3	State3	Event3	31.2

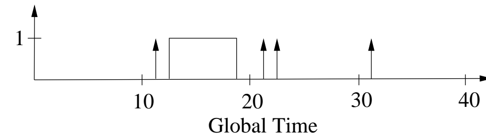

Figure 3. Predicate Value Timeline Example

4. Measure Estimation

One of the main purposes of the Loki fault injector is to assess the dependability and performance of a distributed application. To do this, Loki contains a flexible language to specify 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: at the study level and at 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 two approaches. The first one, called *simple sampling measure*, considers the experiment results of all the studies as similar i.e., as 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*. Before detailing the above three campaign measures, we first describe the measures at the study level.

4.1. Measures Defined at the Study Level

Measures at the study level are based on three concepts: a "predicate," an "observation function," and "subset selection." Each measure is an ordered sequence of (subset selection, predicate, observation function) triples. We now describe in detail these three concepts and their combination to get the study level measures.

4.1.1. Predicate. Predicates in Loki are used to query the global timeline, which was generated during the analysis phase described in Section 2, to identify whether certain conditions are satisfied or not. 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 thus a query 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. 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 [7], 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 timeline thus obtained contains a combination of impulses and steps.

An example of a predicate is `((StateMachine1, State1, 10, 20)|(StateMachine2, State2, Event2, 10, 30)|(StateMachine3, State3, Event3))`. This predicate is true during any time between 10 and 20 ms when StateMachine1 is in State1, at any time between 10 and 30 ms when StateMachine2 is in State2 and Event2 occurs, and whenever StateMachine3 is in State3 and Event3 occurs.

Figure 3 gives an example of a global timeline and shows the predicate value timeline obtained on applying the above predicate 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, we show only the mean of the two time bounds.)

4.1.2. Observation Function. For each defined predicate, the user must specify an *observation function*. The input of an observation function is the predicate value timeline of its predicate: the output is called an *observation function value*. The observation function value extracts the required information from the predicate value timeline as a single value. There are two types of observation functions:

predefined functions and user-defined functions. The predefined observation functions are `count` (which counts the number of times a transition occurs by counting impulses and/or steps), `outcome` (which gives the outcome of the predicate value at a particular instant), `duration` (which measures the length of time the predicate is true or false after a given number of transitions), `instant` (which gives the instant corresponding to a particular transition: the transitions that are considered may be impulses and/or steps), and `total_duration` (which measures the total length of time during which the predicate value is true or false). Note that except for the function `outcome`, the observation functions need to be defined on a specific interval. An example of an observation function is `duration(T,1,10,40)`. This returns the length of time during which the predicate is true ("T") after the first transition from false to true occurred ("1") during the interval [10 ms, 40 ms]. The result on applying this observation function to the predicate value timeline of Figure 3 is 6.5 ms.

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 classical mathematical functions and can be compiled with a standard C compiler.

4.1.3. Subset Selection. As explained earlier, a predicate and an observation function are defined for all experiments in a 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. The user can define a subset function that returns true or false, using classical mathematical functions and observation function values, and can be compiled with a standard C compiler. Selecting all the experiments with a positive observation function value is an example of subset selection.

4.1.4. Combining Predicates, Observation Functions, and Subset Selections. We now explain how to combine predicates, observation functions, and subset selections to obtain measures at the study level.

After defining a predicate and an observation function, the user might want to focus on a subset of experiments in the study based on the observation function values. For this subset, a new predicate and a new observation function can be defined. 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 thus 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 the *final observation value*.

4.2. Measures Defined Across Studies

Final observation values are processed inside each study and across studies to obtain the campaign measure estimation. The campaign measure would 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 [14] (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 on the statistical estimations associated with it.

4.2.1. Simple Sampling Measures. Simple sampling measures in Loki are used when the user does not want to differentiate between the final observation values of different studies. These measures are obtained by considering all the selected studies to be similar such that the final observation 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 value of the jth experiment of study i.

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

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

The three central moments of order 2, 3, and 4 can be obtained from the first four non-central moments by the expressions in [11] p. 18, Eqn. (100):

$$\mu_2 = \mu'_2 - (\mu'_1)^2 \quad (1)$$

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

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

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

$$\beta_1 = \frac{(\mu_3)^2}{(\mu_2)^3} \qquad \beta_2 = \frac{\mu_4}{(\mu_2)^2} \quad (4)$$

Finally, percentiles for various α-levels are obtained by using the Bowman and Shenton approximation [4, 5]. Bowman and Shenton introduced a rational fraction approximation for any percentile y_γ of a standardized distribution ($\mu_1 = 0, \mu_2 = 1$) of the Pearson system. This approximation uses a 19-point formula:

$$y_\gamma = \frac{\pi_{\gamma,1}\left(\sqrt{\beta_1}, \beta_2\right)}{\pi_{\gamma,2}\left(\sqrt{\beta_1}, \beta_2\right)} \qquad \text{where,} \qquad (5)$$

$$\pi_{\gamma,i}\left(\sqrt{\beta_1}, \beta_2\right) = \sum_{0 \le r+s \le 3}\sum a^{(i)}_{\gamma,r,s}\left(\sqrt{\beta_1}\right)^r \beta_2^s \quad \text{for } i = 1, 2 \qquad (6)$$

The values of $a^{(i)}_{\gamma,r,s}$ are given in [4, 5]. When $\mu_3 \ge 0$, the γ-percentile of the non-standardized distribution is given by $z_\gamma = \mu_1 + \sqrt{\mu_2}\, y_\gamma$. When $\mu_3 < 0$, $z_\gamma = \mu_1 - \sqrt{\mu_2}\, y_{(1-\gamma)}$.

4.2.2. Stratified Weighted Measures. Stratified weighted campaign measures are estimated by building samples containing the final observation 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 [3]. When considering several independent studies, the overall coverage of the fault tolerance mechanism is defined by a weighted function across studies [12]. From a statistical viewpoint, the functions for calculating the moments are linear for linearly weighted combinations of the random variables representing the final observation values. Here we assume that the powers of random variables representing the final observation values are independent across studies. Also, 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 4.2.1, we define p_i as the normalized weight associated with the study i. Adapting the previous expressions to focus now on each study, we define the first four non-central moments for study i by:

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

The central moments of order 2, 3, and 4 for study i can be obtained as in Eqns. (1), (2), and (3), with μ_j and μ'_j replaced by $\mu_{j,i}$ and $\mu'_{j,i}$ respectively, for $j = 1, 2, 3, 4$. The mean $\mu'_1 = \sum_{i=1}^{M} p_i \mu'_{1,i}$. Moreover, the central moments of orders 2, 3, and 4 are then given by the following expression:

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

Finally, the skewness and kurtosis coefficients and the percentile points are calculated as explained for the simple sampling measures in Eqns. (4) and (6).

4.2.3. Stratified User Measures. A stratified user measure is a measure in which the final observation 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 for combining final observation values of the various studies. The only result Loki gives in this case is a campaign measure in which each final observation value in the user-defined function is replaced by the mean of the final observation values for that study. However, sometimes the obtained campaign measure value might have no statistical meaning.

5. Leader Election Example

This section focuses on a simple leader election application. We first describe the election protocol and then show how this protocol can be implemented using the new Loki runtime architecture presented in this paper. Finally, we present several measure examples for this leader election protocol.

5.1. Protocol Description

The test application implements a simple leader election protocol (which is an expanded version of that introduced in [7]). The application consists of n processes, each of which runs on a different host. In the protocol, the n processes elect a leader from amongst themselves. To do this, each process chooses a random number and sends it to the remaining $n-1$ processes. The process that chose the highest number is elected as the leader. In case of ties, this arbitration is repeated until it is resolved. When the leader fails by crashing, a new leader is elected through the repetition of this election protocol.

For the test application, all the state machines are identical. This common state machine specification is shown in Figure 4. The nodes in the graph are labeled with state names corresponding to phases of the election application, and the arcs are labeled with local event notifications. At

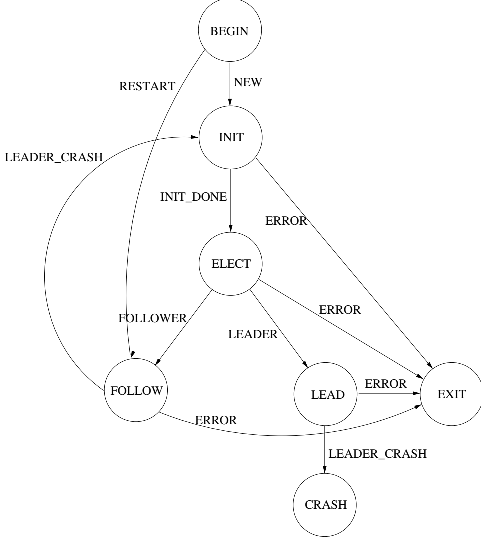

Figure 4. Election Protocol

the start of the application, each state machine is in the BEGIN state. If the state machine is a new one (as indicated by the NEW event), it transitions to the INIT state. If it is a restarted one (as indicated by the RESTART event), it transitions to the FOLLOW state, since a restarted process will always be a follower. The INIT state represents the initialization of the processes, which involves the setting up of communication between the processes. After the initialization, the probe notifies the state machine of an INIT_DONE, and the state machine transitions to an ELECT 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, the probe sends an ERROR notification to the state machine, which then transitions to the EXIT state. When the leader crashes, a LEADER_CRASH notification is generated by the probes of all state machines. The state machine of the leader transitions to the CRASH state and all the other state machines transition to the INIT state, signifying the start of a new leader election.

5.2. Implementation and Instrumentation

Since the test application involves crash of the leader, the previous Loki runtime cannot be used to evaluate this application. The new runtime has to be used for this evaluation. However, the process of instrumenting the application and the process of campaign execution and analysis are the same as given in [6]. The application is instrumented as follows. Since the source code for the application is available, the probe is made a part of the application code. The `injectFault()` method of the probe implements the faults to be injected in the application. Using the `notifyEvent()` method of the state machine, the probe's local event notifications are placed within the application code where the application transitions from one state to another. The `main()` function of the application is renamed `appMain()`. After being instrumented as described above, 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. Then, using the Loki interface as in [6], the campaigns are specified and executed, and the post-runtime analysis is performed.

5.3. Measure Definitions

The goal of the measures presented in this section is to characterize the election process of the leader election protocol. We show that classical fault tolerance measures like coverage (i.e., a measure of whether the leader has been successfully elected in the presence of an injected fault without an error occurring) can be easily defined using the framework presented in this paper. Moreover, we present two performance measures: the number of times a leader is successfully elected within a specified time interval, and the average time taken for a leader election.

The following predicate is used for all the above measures. The predicate is true during the times when any one of the state machines transitions from the ELECT state to the LEAD state (i.e., a successful leader election), or when any one of the state machines is in the BEGIN, INIT, or ELECT states (i.e., an election is in progress). This predicate is applied to the global timeline of each of the experiments to obtain the corresponding predicate value timeline. The obtained predicate value timelines consist of both impulses and steps. Each impulse corresponds to the occurrence of a transition from the ELECT state to the LEAD state in a state machine, i.e., the state machine was successfully elected as a leader. Each step corresponds to any state machine being in either the BEGIN, INIT, or ELECT states, i.e., an election is in progress.

```
((StateMachine1, ELECT, LEADER) | ... |
(StateMachinen, ELECT, LEADER) | (StateMa-
chine1, BEGIN) | ... | (StateMachinen,
BEGIN) | (StateMachine1, INIT) | ... |
(StateMachinen, INIT) | (StateMachine1,
ELECT) | ... | (StateMachinen, ELECT))
```

To estimate the coverage of the election process, we remove the LEADER_CRASH event from the election protocol so that each experiment consists of only one election. The coverage can then be estimated by run-

ning the modified election protocol several times by creating multiple experiments. The observation function for this case is `count(U, I, START_EXP, END_EXP)`, where `START_EXP` and `END_EXP` are Loki keywords that represent the start and end times of an experiment, respectively. The above observation function returns the number of impulses in the predicate value timeline of an experiment. This observation function value is thus 1 if a leader has been successfully elected; otherwise it is 0. Through use of the observation function values of several experiments, the coverage can be statistically estimated.

We now consider the performance measures for whose estimation the election protocol is not modified, i.e., the LEADER_CRASH event is present. Since the LEADER_CRASH event is present, multiple elections might have occurred during each experiment. The observation function for counting the number of times a successful election has taken place in the time interval [0,100] is `count(U, I, 0, 100)`. Also, the observation function for estimating the average time spent electing a leader is `total_duration(T, START_EXP, END_EXP)/count(U, I, START_EXP, END_EXP)`, where `START_EXP` and `END_EXP` have the same meaning as given earlier.

Note that both of the above observation functions, when applied to the predicate value timeline of an experiment, return the observation function value for that experiment. The measures for a campaign are obtained as described in Section 4.

6. Conclusions

This paper presents two new features in Loki that significantly enhance its capabilities. These features are a new runtime and measure estimation. The former version of the Loki runtime did not support dynamic entry and exit of nodes in the system. Therefore, when a node crashed, it was not possible to restart it. This paper presents and compares some design choices for a runtime supporting dynamic entry and exit of nodes, explains why one of them was selected, and describes it in detail. This paper also details a flexible method for estimating a wide range of dependability and performance measures in Loki. The implementation of the new features and the related graphical user interfaces is almost complete. In the future, we will focus on designing several probes that the user of Loki could customize and use in his or her fault injection experiments.

References

[1] 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)*, pages 78–85, May 1997.

[2] 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*, pages 494–501, 1995.

[3] 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*, 20(11):1306–1311, 1971.

[4] K. O. Bowman and L. R. Shenton. Approximate percentage points for pearson distributions. *Biometrika*, 66(1):147–151, 1979.

[5] K. O. Bowman and L. R. Shenton. Further approximate pearson percentage points and cornish-fisher. *Communications in Statistics, Simulation, and Computation*, B8(3):231–244, 1979.

[6] 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)*, pages 237–242, June 2000.

[7] 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*, pages 168–177, October 1999.

[8] 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)*, pages 404–414, June 1996.

[9] K. Echtle 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*, pages 28–35, 1992.

[10] 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*, pages 204–213, 1995.

[11] N. L. Johnson and S. Kotz. *Distibutions in Statistics – Continous Univariate Distributions-1*. John Wiley & Sons, New York, 1969.

[12] 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)*, pages 228–237, 1993.

[13] 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)*, pages 91–100, March 2000.

[14] A. Stuart and J. K. Ord. *Distribution Theory, Kendall's Advanced Theory of Statistics, 1*. Edward Arnold, London, 1987.