A Global-State-Triggered Fault Injector for Distributed System Evaluation *

Ramesh Chandra[+], Ryan M. Lefever[†], Kaustubh Joshi[†], Michel Cukier[‡], and William H. Sanders[†]

[+]Department of Computer Science, Stanford University, Stanford, CA 94301, USA

rameshch@stanford.edu

[†]Coordinated Science Laboratory and Department of Electrical and Computer Engineering

University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

{lefever,joshi1,whs}@crhc.uiuc.edu

[‡]Department of Mechanical Engineering, University of Maryland, College Park, MD 20742, USA

mcukier@eng.umd.edu

Abstract

Validation of the dependability of distributed systems via fault injection is gaining importance, because distributed systems are being increasingly used in environments with high dependability requirements. The fact that distributed systems can fail in subtle ways that depend on the state of multiple parts of the system suggests that a global-state-based fault injection mechanism should be used to validate them. However, global-state-based fault injection is challenging, since it is very difficult in practice to maintain the global state of a distributed system at runtime with minimal intrusion into the system execution. This paper presents Loki, a global-state-based fault injector, which has been designed with the goals of low intrusion, high precision, and high

*This material is based on work supported by NSF under ITR Contract 0086096. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of NSF.

flexibility. Loki achieves these goals by utilizing the ideas of partial view of global state, optimistic synchronization, and offline analysis. In Loki, faults are injected based on a partial view of the global state of the system, and a post-runtime analysis is performed to place events and injections into a single global timeline and to discard experiments with incorrect fault injections. Finally, the experiments with correct fault injections are used to estimate user-specified performance and dependability measures. A flexible measure language has been designed that facilitates the specification of a wide range of measures.

Index Terms: Distributed Systems, Reliable Systems, System Evaluation, Fault Injection, Partial View of Global State, Off-line Clock Synchronization, Measure Estimation.

1 Introduction

The increasing use of distributed systems in applications with high availability and reliability requirements, ranging from commercial web servers to air traffic control systems, makes it necessary to develop techniques to validate the dependability of these systems. Fault injection has been among the most practical and effective of these techniques, and can be defined as a way to test a fault-tolerant system with respect to a class of inputs specific to such a system, i.e., the faults [2].

Since failures in a distributed system can depend on its global state, it is important that fault injections in a distributed system be triggered based on its global state. Global-state-based triggers are useful for both fault removal and system evaluation. Global-state-based triggers are useful for fault removal because they make it possible to introduce faults at very precise points in the execution of the system and observe the effects in detail. Since system evaluation requires the fault injection profile to be representative of reality, it may not be obvious why sophisticated global-state-based triggers are needed for evaluation. However, there are three important reasons why such triggers are useful. First, since global-state-based triggers can be used to drive a distributed system into certain global states that can be hard to reach otherwise, they are useful in exercising and evaluating hard-to-reach parts of the system. Second, distributed systems are evolving to the point that both the fault models and the measures needed to describe their dependability properties are dependent on their state. For example, software faults are part of an increasingly important class of faults that may need

certain preconditions in order to manifest themselves. Global triggers can be used both to drive the system into the states where these preconditions are true, and to inject faults only when they are. Third, global-state-based fault triggers can be used to obtain conditional system measures, which can then be used with models to get system measures for large classes of workloads.

Global-state-based fault injection of a distributed system is inherently difficult, because individual nodes in the system may not be aware of the current global state of the entire system at any point in time. Past research in fault injection has focused on stand-alone systems (e.g., [21, 20, 24]), distributed systems with local-state-based triggers (e.g., [12, 9, 22]), and centralized simulation of faults in distributed systems [1]; very little work has been done in global-state-based fault injection of distributed systems. Existing work on determining the global state of a distributed system includes techniques such as snapshot algorithms [7], distributed debugging [18], and distributed monitoring [16, 4]. Those methods log the local states of nodes in the system for post-processing, and are useful for performance monitoring or fault monitoring. Because of the offline nature of those techniques, it is not possible to create the required fault injector simply by extending them with triggering mechanisms so that they can inject faults. Similarly, synchronizing the system at every state change and performing the required fault injections will not work, since it is very intrusive to system execution and could change the behavior of the system in an undesired way. Thus, a new mechanism is needed that will perform global-state-based fault injection in distributed systems.

Loki is a distributed system fault injector that has been developed to fill this need. We believe that the key requirements for a fault injector are low intrusion, high precision, and high flexibility; Loki has been designed to meet these requirements in the context of global-state-based fault injection. A few key ideas have shaped Loki's design so as to make this possible. Those ideas include the concepts of (a) state machine abstraction, (b) optimistic synchronization, and (c) separation of mechanism from policy in fault injection, as well as the observations that (d) fault triggers depend only on a part of the global state and that (e) offline analysis is sufficient for system evaluation.

Items (b), (d), and (e) help in decreasing the intrusion and increasing the precision of fault injection using Loki. Observation (d) implies that, in general, the triggers for fault injections in a

particular node depend on the state of only a few other nodes. Loki makes use of this observation to optimize the amount of state update traffic on the network by sending state change notifications only between the required nodes, and by tracking only the required part of the global state at each of the nodes. Observation (e) suggests that since fault injection is a process of evaluation, it does not need online synchronization. Loki utilizes this observation and performs offline analysis, thus decreasing the runtime intrusion. With respect to (b), while checking fault triggers and performing fault injections, Loki optimistically assumes that the nodes are correctly synchronized due to the state change notifications, i.e., the locally available view of the global state is assumed to be right. However, this assumption could result in fault injections in the wrong global states. We remedy that during the offline analysis by verifying each fault injection and discarding experiments with incorrect fault injections. This loose coupling among the components of Loki decreases its runtime intrusion and improves its scalability; thus, Loki performs well as long as the relevant state changes are not too rapid. In addition, Loki uses hardware clocks whenever possible to further decrease runtime intrusion [13] and record experiment data with precision.

In the past, many fault injectors have been built for specific systems, and it was non-trivial to extend them to evaluate other systems. In contrast, Loki has been built to be flexible so that it can be used for evaluating different kinds of systems with varying fault types. This has been achieved through use of item (c), i.e., separation of the mechanism used to carry out a fault injection (namely, application-independent tasks such as running of experiments, collection of measurements, clock synchronization, analysis of results, and obtaining of measures) from the policy of fault injection (namely, application-specific tasks such as state machine specification, types of faults, and specification of fault triggers). Additionally, Loki provides flexibility in the kinds of measures that the user can obtain from experiment data. Most previous fault injectors do not provide such flexible means to process the data collected. An extensive graphical user interface has been developed for Loki to assist the user in the specification of fault injection campaigns, execution of the campaigns, and obtaining desired measures from the results of the campaigns. In addition to being the management console for the Loki fault injector, the graphical interface performs important runtime functions. A description

of Loki's graphical user interface and other operational details of the Loki tool are described in the attached Appendix A.

In this paper we present proof-of-concept results from experiments in which Loki was used to inject faults in a simple leader election protocol. The results verify the Loki fault injection tool's functionality. However, Loki has also been used in two case studies involving experimental evaluation of large-scale distributed systems. The first case study is an experimental evaluation of unavailability due to the group membership protocol in the Ensemble group communication system [14]. The second case study is an experimental evaluation of some of the protocols used to provide high availability in the Coda distributed file system [17]. Both case studies demonstrate the efficacy of Loki in evaluating complex distributed systems.

2 Loki Concepts and Terminology

- *State and state machine specification:* The concept of state is fundamental to Loki. The user has to choose the abstraction of states for each component of the distributed system and define a state machine to track the component's state at that level of abstraction. The execution of the component generates *local events* that trigger transitions in the state machine. The current state of any component is called its *local state*, and the vector of the local states of all the components in the system is the *global state*.

- *Node:* The user should divide his/her distributed application into a set of basic components, each of which has a *state machine specification* and a *fault trigger specification*. Each of these basic components, combined with the Loki runtime code, is called a *node*. All fault injections and recordings of the state change and fault injection times are performed at the node level.

- *Fault trigger:* A fault trigger specifies the conditions under which a particular fault is to be injected into a node. It is defined as a Boolean expression over the global state of the system under study. Variables of the expression are represented by (state machine, state) pairs. A fault is injected into the node if the fault trigger transitions from a false to a true because of a global

state change. An example trigger is `((SM1:ELECT)&(SM2:FOLLOW))`, which indicates that the corresponding fault is to be injected when the system transitions from a global state in which the trigger is not satisfied to a global state in which the state machine `SM1` is in state `ELECT` and state machine `SM2` is in state `FOLLOW`. Note that Loki does not place any restriction on the type of fault (e.g., random bit-flip or message corruption). This gives the user considerable flexibility in choosing the faults best suited for the system under study.

- *Partial view of global state:* To inject faults into a node based on the node's fault triggers, it is sufficient to track the "interesting" portion of the global state that is necessary for fault injection at that node, i.e., the vector of the local states of the nodes on which the fault triggers depend. This interesting portion is called the *partial view of global state* at that node.

- *Fault injection campaign, study, and experiment:* The process of fault injection of a distributed system using Loki consists of one or more fault injection campaigns. Each campaign consists of one or more studies. For each study, the user defines the nodes for his/her system. Although the division of the fault injection process into campaigns and studies is left to the user's discretion, it is desirable for a study to consist of a set of similar fault injections and for a campaign to consist of a set of correlated studies. That is so the user can obtain meaningful results from his/her experiments using Loki's measure estimator, which is described in Section 5. To obtain statistically accurate measures, several runs of each study are performed along with the associated fault injections. Each of these runs is called an *experiment*.

- *Fault Injection Process:* Loki's fault injection process comprises a specification process and a campaign evaluation process. During the specification process, the user provides the system's state abstraction, fault details, and measures to be computed, and prepares the system under study for fault injection. The campaign evaluation process then makes use of this specification. The campaign evaluation process is subdivided into a runtime phase to execute the fault injection campaign, an analysis phase to check for proper fault injections, and a measure estimation phase to obtain desired measures from global timelines created in the analysis phase.

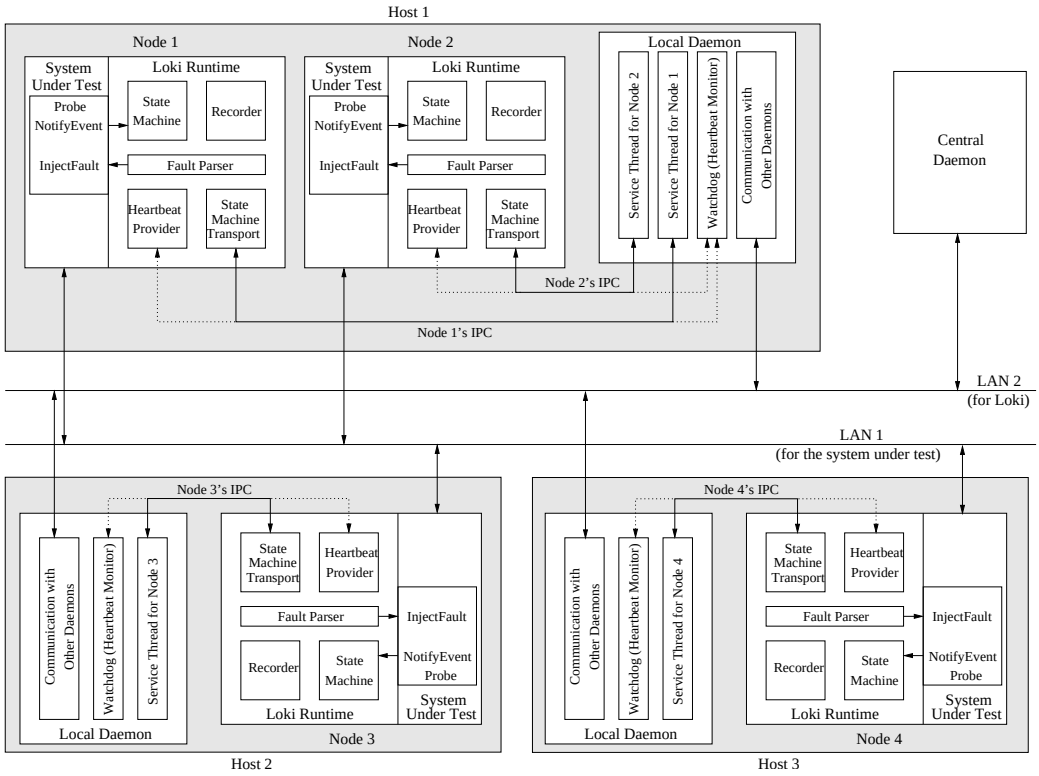


Figure 1: The Loki Runtime Architecture

3 The Runtime Phase

During the runtime phase, fault injection campaigns are executed. The runtime phase consists of executing the specifi ed number of experiments in every study in a campaign, performing the required fault injections, and recording the required experiment data. The Loki runtime is the module of the Loki fault injection tool that manages the runtime phase of the fault injection campaigns.

The Loki runtime architecture is illustrated in Figure 1, with four nodes on three hosts. In this architecture, one portion of the Loki runtime is attached to each of the nodes in the system; the node's *runtime* performs fault-injection-related operations. Each node's runtime consists of several components, namely the *state machine, state machine transport, parser, probe, recorder,* and *heartbeat provider*. All of the components except the probe are independent of the system under study; the user is responsible for the probe implementation. The runtime also contains one *local daemon* per host and one *central daemon* for the entire runtime system. The daemons keep track of the dynamic information regarding the current nodes in the system, and facilitate routing of messages between

nodes even when nodes are dynamically exiting and entering the system.

The main function of a state machine[1] is to keep track of the partial view of global state necessary for fault injection into its node. Even if multiple nodes share the same executable code, each node has a separate state machine with a unique name. To keep track of the partial view of global state of a node, the node's state machine has to track both its local state and the state of the required remote nodes. To keep track of the local state, the state machine uses the state machine specification provided by the user along with the node's local event notifications. When the state machine receives a local event notification, it computes its new state based on both its old state and the local event. To keep track of the state of remote nodes, the state machines send only the required state change notifications to each other, in accordance with the specification. The ideas of maintaining only the required partial view of global state and transmitting only the required state change notifications both help to decrease the intrusion to the application.

The state machines use their respective state machine transports to send notification messages to each other. The state machine transports abstract the underlying communication system, and provide a simple interface for sending and receiving notification messages. To do so, the state machine transports maintain communication routes to their respective local daemons.

On every change in the partial view of global state of a node, its state machine notifies its fault parser. The fault parser uses the partial view of global state to check whether any local fault triggers have transitioned from false to true. If they have, the parser instructs its probe to inject the corresponding faults.

The probe is the only system-dependent component of the Loki runtime, and performs two main functions: notifying its state machine of any local events, and performing the actual fault injection into the node when instructed to do so by the fault parser. Since the probe is system-dependent, the user has to implement it as part of the instrumentation of the distributed application, as discussed in Section A.2.2 of the Appendix.

The function of a node's recorder is to record information regarding local state changes and fault

[1]The implementation of the system-independent state machine component of the runtime is referred to as the *state machine*, while the system-dependent description of the state machine is referred to as the *state machine specification*.

injections, along with their occurrence times, to a local timeline. To improve precision and decrease intrusion, the recorder uses hardware clocks [13]. Finally, the heartbeat provider sends “I am alive” messages to its node’s local daemon. This aids the local daemon in detecting crashes.

The main functions of the local daemon are to co-ordinate the experiment execution on its host, start new nodes at the beginning of an experiment, route the notification messages between hosts, monitor the nodes on its host for crashes, and manage node crashes and restarts. To perform these operations, the local daemons coordinate with each other and with the central daemon using reliable, ordered communication links (currently TCP). Each local daemon communicates with the nodes on its host using IPC (currently shared memory with semaphores).

The central daemon manages the runtime phase of the campaigns. That phase includes starting up all the components required for each experiment, aborting experiments if there are abnormal situations (such as local daemon crashes or experiment hangs), and determining when experiments have completed. The central daemon coordinates with the local daemons to perform these functions.

Nodes, local daemons, and the central daemon are all single processes. Though a node consists of many components, it has only three Loki runtime threads in addition to the application threads. The three runtime threads are the state machine transport thread, parser thread, and heartbeat thread. The state machine component of the runtime executes in the application threads. The Loki front end functions as the central daemon; for more details regarding its implementation, refer to Section A.2 of the Appendix. A detailed description of the operation of the runtime can be found in Section A.1 of the Appendix.

4 The Analysis Phase

By the time the runtime phase of the fault injection campaign has completed, a large amount of timeline data has been recorded during the running of experiments. This data has to be processed before meaningful conclusions can be drawn about the system under study. The processing consists of two phases: an analysis phase and a measure estimation phase. The analysis phase, described in this section, consists of two steps, namely the conversion of local timelines in each experiment into

a single global timeline for that experiment, and the verification of the correctness of fault injections in each experiment. Experiments in which fault injections occurred in incorrect global states are discarded. Note that "analysis" in this context refers only to the process of computing a correct global timeline from raw measurement data. It does not include the analysis of the application behavior or detected faults; that analysis can be done by the user based on the estimated measures and global timeline after the measure estimation phase.

4.1 Conversion to Global Timeline

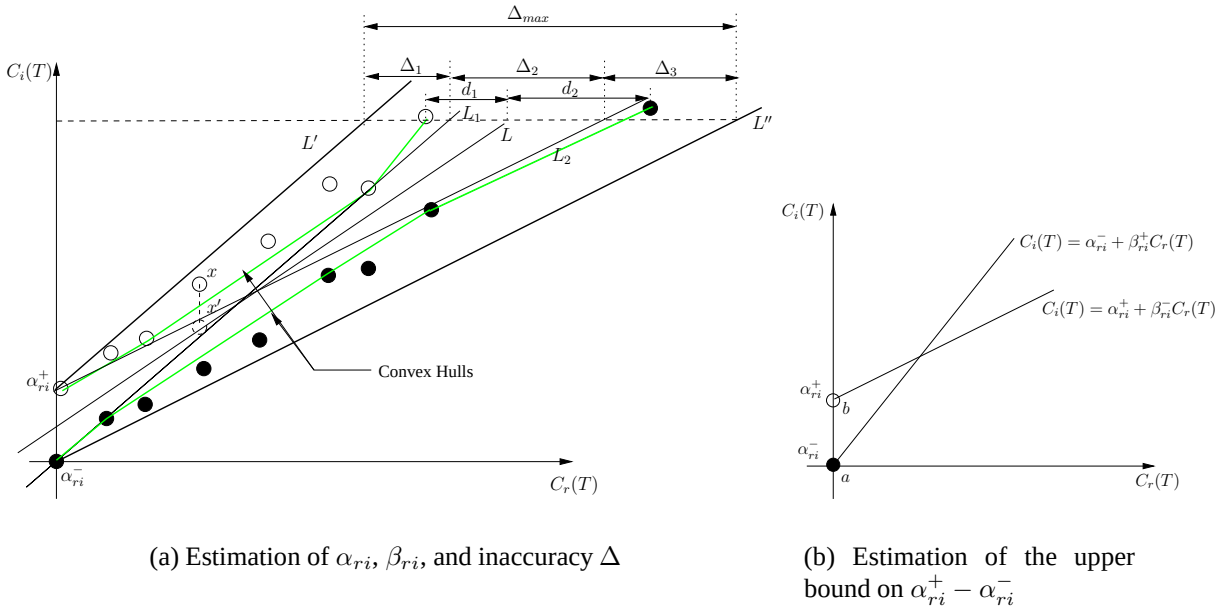
Loki uses an offline clock synchronization algorithm to calibrate the clocks on the multiple machines on which the fault injector operates, so that all the local timelines of an experiment can be combined into a single global timeline [13]. One machine's clock is taken as the reference, and the offset and drift rate of every other machine's clock are estimated relative to the reference clock. These offsets and drift rates are then used to place all the local times onto a single global timeline.

Loki assumes that the drifts of the processor clocks of the different machines in the distributed system are linear [11]. 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:

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

where α_{ij} is the offset between the clocks of machine i and machine j at $t = 0$, and β_{ij} is the drift (or skew) 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 α_{ri} and β_{ri}, for $i = 1, \ldots, m$. (It can be easily seen that $\alpha_{rr} = 0$ and $\beta_{rr} = 1$.) To compute these values, synchronization messages are passed between the reference machine and all the other machines, before and after each experiment or study, during the runtime phase. During this process, local timestamps are recorded for each message at the sender and receiver. The synchronization messages are passed between, not during, experiments, so that the



(a) Estimation of α_{ri}, β_{ri}, and inaccuracy Δ

(b) Estimation of the upper bound on $\alpha_{ri}^+ - \alpha_{ri}^-$

Figure 2: Estimation of Parameters in the Analysis Phase

intrusion into application execution is reduced.

Figure 2(a) gives a graphical illustration of these timestamps for hypothetical messages between machine i and machine r. The x-axis is the local time, $C_r(t)$, on machine r, and the y-axis, $C_i(t)$, is the local time on machine i. Each point represents a message, whose coordinates are the local send or receive timestamps at machines r and i. Note that since we are interested only in the times of occurrences of events relative to each other, the timeline axes can be translated as desired. In particular, the axes can be translated so that the point corresponding to the first synchronization message falls on either of the axes or on both. The line $L \equiv C_i(t) = \alpha_{ri} + \beta_{ri} C_r(t)$ is the exact relation between local times on machine r and machine i. If the message delays in the network are zero, then all the points would lie on L. However, since in practice message delays are not zero, the points are separated from L by a distance equal to their message delay. All the points above L (represented by unfilled circles) represent messages from machine r to machine i, and the points below L (represented by filled circles) represent messages from machine i to machine r. So, for example, in the absence of delays, the message represented by point x would actually have been at x', and the transmission delay from machine r to i for this message is the vertical distance between

x and x'.

The question is how to determine L given only the points (representing messages) in the figure. If the message delays are fixed (or known in some other way), then it is a trivial matter to compute L from the points. However, in a general distributed system, message delays are variable, so the exact determination of L becomes impossible. It is, however, possible to estimate bounds on L, i.e., to estimate the bounds on α_{ri} ($[\alpha_{ri}^-, \alpha_{ri}^+]$) and β_{ri} ($[\beta_{ri}^-, \beta_{ri}^+]$). To do so, observe that $L_1 \equiv C_i(t) = \alpha_{ri}^- + \beta_{ri}^+ C_r(t)$ and $L_2 \equiv C_i(t) = \alpha_{ri}^+ + \beta_{ri}^- C_r(t)$ are the two extreme estimates of L, because they satisfy L's property of dividing the points represented by filled and unfilled circles. Therefore, Loki uses L_1 and L_2 to compute the lower and upper bounds of α_{ri} and β_{ri}, namely $\alpha_{ri}^-, \alpha_{ri}^+, \beta_{ri}^-$, and β_{ri}^+. L_1 and L_2 themselves are computed by first taking the convex hulls for the upper and lower sets of points and then joining their "edge" points, as illustrated in Figure 2(a).

Further details regarding Loki's analysis algorithm can be found in [13]. Note that $[\alpha_{ri}^-, \alpha_{ri}^+]$ and $[\beta_{ri}^-, \beta_{ri}^+]$ are not confidence intervals for α_{ri} and β_{ri}. With confidence intervals, a value has only a high probability of being in a certain interval, but in this situation, the correct values of α_{ri} and β_{ri} are always in the intervals $[\alpha_{ri}^-, \alpha_{ri}^+]$ and $[\beta_{ri}^-, \beta_{ri}^+]$, respectively (even though their exact values are unknown). Methods for increasing the accuracy of the estimates of α_{ri} and β_{ri} include increasing the number of synchronization messages, increasing the duration of synchronization messages, and increasing the duration of each experiment by adding delays before and after the experiment.

As mentioned in Section 3, the occurrence times of 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 node's 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 onto a single (reference) timeline. This is done as follows: suppose an event occurred on machine i at local time $C_i(T)$. Then, from Eqn. (1), we have the reference clock time as $C_r(T) = \frac{C_i(T) - \alpha_{ri}}{\beta_{ri}}$. However, since only the bounds (and not the exact values) for α_{ri} and β_{ri} are known, only the upper and lower bounds of $C_r(T)$ can be found using $C_r(T)^- = \frac{C_i(T) - \alpha_{ri}^+}{\beta_{ri}^+}$ and $C_r(T)^+ = \frac{C_i(T) - \alpha_{ri}^-}{\beta_{ri}^-}$.

Therefore, an event occurring at local time $C_i(T)$ on machine i corresponds to an event occurring

between time bounds $C_r(T)^-$ and $C_r(T)^+$ on the reference machine r. Using this method, Loki projects all the events in the local timeline of all the nodes in the system onto a (reference) global timeline. Graphically, all the $C_r(T)^-$ fall on the line L', and all the $C_r(T)^+$ fall on the line L'', where L' and L'' are as shown in Figure 2(a). L' is parallel to L_1 and L'' is parallel to L_2, with $L' \equiv C_i(t) = \alpha_{ri}^+ + \beta_{ri}^+ C_r(t)^-$, and $L'' \equiv C_i(t) = \alpha_{ri}^- + \beta_{ri}^- C_r(t)^+$.

A valid point of concern here is Δ, the size of the interval $[C_r(T)^-, C_r(T)^+]$; Δ is the inaccuracy in the computed global time, and determines the precision of all events in the system, including fault injections. In Figure 2(a), for any local time $C_i(t)$, Δ is the horizontal distance between lines L' and L'' at $C_i(t)$. Obviously, it is desirable that Δ be as small as possible. It can be seen from the figure that Δ increases linearly with increasing $C_r(t)$. The maximum inaccuracy, Δ_{max}, occurs at the end of a study. As illustrated in the figure, Δ_{max} can be decomposed into $\Delta_1, \Delta_2,$ and Δ_3, where $\Delta_1 = \frac{\alpha_{ri}^+ - \alpha_{ri}^-}{\beta_{ri}^+}, \Delta_3 = \frac{\alpha_{ri}^+ - \alpha_{ri}^-}{\beta_{ri}^-}$., and Δ_2 is less than twice the maximum message delay (as measured in global time). Without loss of generality, it can be assumed that the last filled circle has a larger $C_i(t)$ than the last unfilled circle, and that Δ_2 is measured along the horizontal line passing through the last unfilled circle (as shown in Figure 2(a)). Then it is easy to see that the delays of messages corresponding to the last unfilled and filled circles (d_1 and d_2 respectively) are less than the maximum message delays, and hence Δ_2 is less than twice the maximum message delay. Figure 2(b) shows the first messages during a synchronization phase. As shown in the figure, we ensure in our implementation that machine i sends the first synchronization message a to the reference machine, and the reference machine sends its first message b immediately after it receives a. Hence it can be observed that $(\alpha_{ri}^+ - \alpha_{ri}^-)$ is bounded by the maximum message delay. Since both β_{ri}^- and β_{ri}^+ are constants (because of the linear drift assumption), this means that both Δ_1 and Δ_3 are bounded by a constant factor of maximum message delay. Hence Δ_{max} is bounded by a constant factor of the maximum message delay. Also, it is possible to decrease Δ by increasing the accuracy of estimates on α_{ri} and β_{ri} as described above, and by decreasing message delay (which increases accuracy of α_{ri}).

4.2 Verification of Fault Injections

After the conversion to the global timeline, the fault injections are checked to determine whether they were proper, i.e., they occurred in the correct global state as specified in the fault specification. For the purposes of this discussion we refer to the event that caused a fault injection predicate to become true as the *entry event* for the corresponding fault, and the event that caused the fault injection predicate to become false as the *exit event* for the fault.

Improper fault injections are removed via a check that determines 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 fault's entry event and exit event. More specifically, the upper bound of the entry event 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 exit event 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. This procedure is repeated for each injection that should have been made in the experiment; the experiment is marked as successful only if all the injections in the experiment were done correctly. The unsuccessful experiments are discarded and not used for measure computation.

5 The Measure Estimation Phase

Measure estimation is a key component of performance and dependability assessment using fault injection. The measures to be obtained from a fault injection study 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, any mechanism for measure estimation in fault injection should be flexible. To this end, a flexible mechanism for measure estimation has been developed in Loki.

5.1 Overview of Measure Estimation in Loki

Measure estimation consists of two steps: *measure specification* and *measure computation*. During measure specification the user specifies the measures he/she wants to compute, and Loki uses these

specifications to perform the measure computation. Loki provides the user with a flexible measure specification language, and uses statistical techniques to compute estimators from the data collected.

Measures in Loki are at two levels: the *study level* and the *campaign level*. As mentioned in Section 2, a study consists of similar fault injections, and a campaign generally consists of correlated studies. Measure estimation at both levels consists of both measure specification and measure computation. A study-level measure is specified for a particular study and is computed from the data collected during execution of the experiments in that study. A campaign-level measure is specified for a particular campaign and is computed by combining study-level measures of a subset of its studies. The measures language is split into two levels to allow computation of measures, such as fault coverage, that may involve several types of faults. For example, a user can conduct multiple studies (one for each fault type), define study-level measures on each study to compute coverage for its fault type, and then combine the results via a campaign-level measure to compute overall coverage for several probable fault type distributions.

5.2 Study-Level Measures

A study-level measure is computed using the global timelines of the experiments in the corresponding study. Informally, a study-level measure consists of (a) a sequence of filters that eliminate all the experiments of the study whose global timelines do not satisfy certain conditions, and (b) the computation of a function using the global timelines of the remaining experiments, whose output is a sample for the study-level measure. More formally, a study-level measure is based on the concepts of *predicate*, *observation function*, and *subset selection*. The above three concepts may be described as follows:

- *Predicate*: Predicates in Loki are Boolean expressions used to test the global timeline to identify whether certain conditions are satisfied by querying the different attributes of the state machines (i.e., states, events, and times), and are either true or false as a function of time. Predicates combine state machine queries with the Boolean operators. Queries can test for the occurrence of a specific state or a specific event in a particular machine in specified intervals of

Global Timeline			
State Machine	Begin State	Event	Time
StateMachine1	State0	Event1	12.4
StateMachine1	State1	Event2	18.9
StateMachine3	State3	Event3	22.3
StateMachine1	State4	Event4	26.3
StateMachine2	State0	Event5	32.3
StateMachine2	State2	Event6	35.6

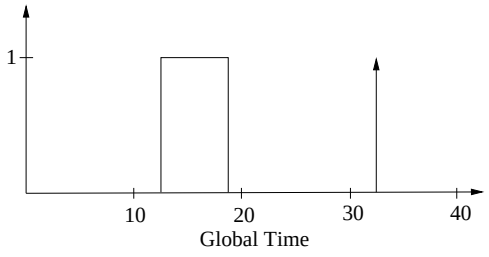


Figure 3: Predicate Value Timeline Example

time. An example of a predicate is `((StateMachine1, State1, 10 < t < 20) | (StateMachine2, Event5, 30 < t < 40))`. This predicate is only true during any time between 10 and 20 ms when StateMachine1 is in State1, and during any time between 30 and 40 ms when Event5 occurs in StateMachine2. 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 results in a Boolean-valued function of time called the *predicate value timeline*. The obtained predicate value timeline contains a combination of impulses and steps. Figure 3 gives an example of a global timeline and shows the predicate value timeline obtained upon application of the above predicate to the global timeline. For more examples of predicates, refer to Section 6.1.

- *Observation function*: For each defined predicate, the user must specify an observation function. The observation function is used to extract a single value from the predicate value timeline. The input to an observation function is a predicate value timeline, and the output is an *observation function value*. Loki provides the functions `count`, `outcome`, `duration`, `instant`, and `total_duration`, which extract information about the predicate value timeline. The observation function can be specified as any C function that uses the above functions along with any mathematical functions and returns a numerical value. An example of an observation function is `total_duration(True, 10, 35)`. This returns the total duration for which the predicate timeline was true between 10 and 35 ms. Its value for the predicate value timeline of Figure 3 is 6.5. For more examples of observation functions, refer to Section 6.1.

- *Subset selection*: After obtaining the previous 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. The first subset selection is predefined to include all experiments. An example of a subset selection is all the experiments with positive observation function values.

Formally, a study-level measure is specified as an ordered sequence of tuples: ((subset_selection$_1$, predicate$_1$, observation_function$_1$), . . ., (subset_selection$_n$, predicate$_n$, observation_function$_n$)), where $n \geq 1$. The subset_selection$_1$ function is always true. A study-level measure is computed as follows. Since subset_selection$_1$ is always true, predicate$_1$ and observation_function$_1$ are computed on the global timelines of all the experiments in the study. The output value of observation_function$_1$ for each of these experiments is checked to determine whether it satisfies the subset_selection$_2$. If it does not, then the experiment is eliminated from further consideration. The global timelines of the experiments that survive this filtering are used to compute predicate$_2$ and observation_function$_2$. This process of filtering is continued for all the tuples in the sequence, when the global timelines of the unfiltered experiments are used to compute predicate$_n$ and observation_function$_n$. The output of observation_function$_n$ for each remaining experiment, is called the *final observation function value* for that experiment, and is a sample for the study-level measure.

5.3 Campaign-Level Measures

Loki processes final observation function values to obtain a study-level measure, and combines study-level measures 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 is theoretically possible to

exhibit all the properties of the distribution in terms of the moments [23] (pp. 108-109). In practice, the properties obtained when calculating the first four moments are very close to the properties of the real distribution. That is the approach that Loki takes. There are three types of campaign measures, which are described below.

Simple Sampling Measures are used when the user does not want to differentiate among the final observation function values of different study-level measures. Simple sampling measures are obtained by considering all the selected study-level measures to be similar such that all of their final observation function values are contained in a single sample, i.e., they are all instances of the same random variable.

Stratified Weighted Measures are used when study-level measures need to be combined using a user-specified linear weighted function, e.g., $ax + by$. The moments for such measures are determined by computing the moments for each constituent study-level measure, weighting them using the user-defined weights, and summing them. This computation assumes that the powers of random variables representing the final observation function values are independent across studies. There are several reasons for focusing on stratified weighted measures when evaluating fault tolerance mechanisms. For example, one important parameter is the coverage of a fault tolerance mechanism [5]. When several independent studies are being considered, the overall coverage of the fault tolerance mechanism is defined by a linear weighted function across studies [19].

Stratified User Measures are used when study-level measures need to be combined using a user-defined function other than a linearly weighted function. Unfortunately, statistical features are not computed for these measures, since the calculation of the first four moments is not a trivial task for any arbitrary function that combines final observation function values of the various studies. Hence, Loki does not combine final observation function values individually, but rather combines the means of the final observation function values for each study-level measure.

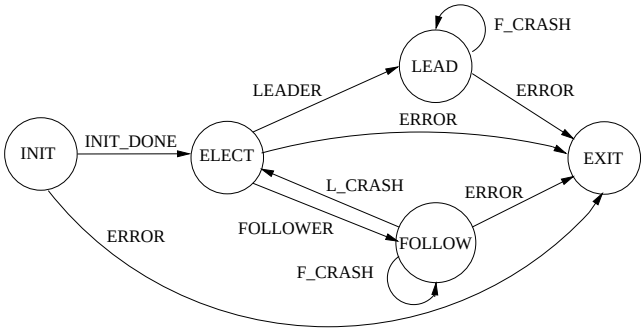


Figure 4: State Machine Specification for a Simple Leader Election Protocol

For the simple sampling and stratified weighted measures, the important statistical properties, such as the first four moments, the skewness coefficient, the kurtosis coefficient, and the percentiles for various α-levels, are computed by Loki as described in [6]. For stratified user measures, only the single value returned by the user-defined function is computed.

6 Sample Application: A Simple Election Protocol

In this section, we describe the instrumentation of a sample distributed application, as a proof-of-concept of Loki's capabilities. Loki has also been used in two case studies involving experimental evaluation of complex distributed systems [14, 17]. The sample application, described here, is a simple leader election protocol, in which n processes elect a leader from among themselves. To do so, 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, the arbitration is repeated until a leader is selected. Process failures are detected by other processes via socket connection closure. If the leader fails, then the remaining processes re-elect a new leader using the same protocol but with one fewer node. In this simple protocol, we assume that only one node fails at a time and that a leader re-election is completed between two successive node failures.

For this application, all nodes use the same state machine specification, which is shown in Figure 4. The nodes in the state machine are labeled with state names corresponding to the phases of the application, and the edges are labeled with transition names corresponding to local event notifications. We use a simple crash failure fault model to illustrate the fault-triggering and measure

Node	Fault Name	Fault Expression
corvette	cleadfault	(corvette:LEAD)
corvette	celectfault	(nsx:ELECT) & (corvette:ELECT) & (xj220:ELECT)
nsx	nleadfault	(nsx:LEAD)
nsx	nelectfault	(nsx:ELECT) & (corvette:ELECT) & (xj220:ELECT)
xj220	xleadfault	(xj220:LEAD)

Table 1: Fault Expressions for the Simple Leader Election Protocol

estimation capabilities of Loki, and to measure Loki's runtime efficiency. We implemented this fault model simply by writing to a null pointer dereference, thus causing a crash. However, as mentioned in Section 2, the fault model supported by Loki is flexible and can be arbitrarily complex depending on the application and type of evaluation. The fault triggers for the example are shown in Table 1.

6.1 Measures

Using Loki's measure language, we were able to compute several interesting measures for the election protocol application. These measures are shown below, along with their definitions in the Loki measure language. Complete details of the functions used in the measure language are given in [6]. For the purposes of the example, the functions `total_duration(TRUE, t1, t2)` and `count(UP, STEP, t1, t2)` return the amount of time the predicate timeline was `TRUE` and the number of low-to-high step transitions on the predicate timeline between times `t1` and `t2`, respectively. `START_TIME` and `END_TIME` refer to the experiment start time and experiment end time, respectively. Tuples for the predicate timelines are shown in the format `(statemachine, state)` or `(statemachine, event)`, as appropriate.

- *Availability* for this application can be defined as the fraction of the total execution time for which there is a functional leader (i.e., leader is in the `LEAD` state). It can be defined in Loki via a single tuple with the predicate function `(nsx, LEAD)|(corvette, LEAD)|(xj220, LEAD)` and the observation function `total_duration(TRUE, START_TIME, END_TIME) / (END_TIME - START_TIME)`.

- An important global state in this application is the one in which any surviving node is in the

ELECT state. Since we assume that crashes do not occur when an election is in progress, such a state can be expressed using the following predicate function: `(nsx, ELECT)|(corvette, ELECT)|(xj220, ELECT)`.

Based on that, a few interesting measures can be computed. In particular, we can compute the number of elections per run by counting the number of times this global state was entered; the observation function is `count(UP, STEP, START_TIME, END_TIME)`.

We can also compute the amount of time spent per election by dividing the time spent by the system in this global state by the number of times this global state was entered. The observation function for this is `total_duration(TRUE, START_TIME, END_TIME)/ count(UP, STEP, START_TIME, END_TIME)`.

- We can also compute the total number of crashes that occurred. To do so, we can use the fact that Loki automatically detects a node crash and generates a special CRASH event that corresponds to the crash in the timeline. We can simply count the number of these crash events to get the required measure. The counting is done via a single tuple consisting of the predicate function `(nsx, CRASH)|(corvette, CRASH)|(xj220, CRASH)` and the observation function `count(UP, IMPULSE, START_TIME, END_TIME)`.

6.2 Performance Evaluation

Two important metrics for evaluating the performance of a tool such as Loki are *intrusiveness* and *injection efficiency*. Intrusiveness is the amount of perturbation in the application's execution caused by the process of fault injection. Injection effi ciency is the probability of correct fault injection (i.e., in the correct global state) by Loki under various conditions. It is important because it ultimately determines the granularity at which faults can be injected. We present some results regarding these metrics in the following sections. Note that the results presented below only measure the overhead of the Loki runtime (i.e., intrusiveness specifi c to global-state-based fault triggering). Any probe-specifi c or fault-model-specifi c intrusiveness would be in addition to the intrusiveness results presented below.

6.2.1 Injection Efficiency

In order to understand the factors on which the injection efficiency of Loki depends, we make the following observation. As described in Section 4.2, fault injections must be discarded by the post-runtime analysis only if the bounds for the fault injection event overlap the bounds for the entry event or the exit event. The probability that such overlap will occur depends primarily on two factors.

The first factor is the precision of the fault injection mechanism, where *precision* refers to the time between the entry event and the actual fault injection. Although a lower numerical value for precision indicates a responsive injection mechanism, it increases the probability that experiments containing correctly injected faults will be thrown away by the post-runtime analysis. To understand why, we define the interval between the upper and lower bounds associated with each event on the global timeline as the *uncertainty interval*. We cannot resolve the time of event occurrence to be any narrower than this interval. The lower the numerical value of the precision of the fault injector, the closer the fault injection event will be to the entry event, and the higher the probability that the injection will be discarded because the uncertainty intervals for these two events overlap. The probability also increases with the length of the uncertainty intervals. To improve the injection efficiency, it is possible to decrease the uncertainty interval by decreasing the value of Δ in the analysis phase, as explained in Section 4.1.

The second factor that influences the injection efficiency is the overlap of the injection event with the bounds for the exit event. The amount of time spent by the system in the global state in which the injection is to occur determines whether such an overlap occurs. We refer to this time as the *state-holding time*. Clearly, the smaller the state-holding time, the higher the probability that an overlap will occur and the fault will be injected incorrectly. It is important to note that the holding time required for successful injection depends on both the precision of the fault injection mechanism and the uncertainty interval (the longer the uncertainty interval, the higher the minimum holding time). However, one important difference between the first factor and the second is that the minimum holding time requirement affects the injection efficiency only for short-lived states, while the overlap between uncertainty intervals for the fault injection event and the entry event interval

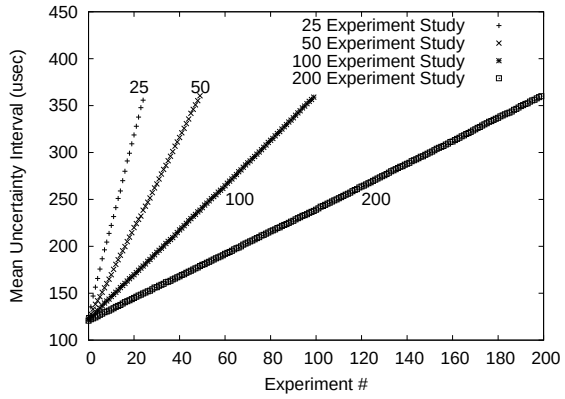
affects the efficiency of *all* fault injections.

To measure the uncertainty interval, we ran four studies of the election protocol, each with 25, 50, 100, and 200 experiments respectively. The runs were performed on 1GHz Pentium III machines running Linux connected to a switched 100 Mbps Ethernet network. Figure 5(a) shows the uncertainty interval for the experiments in each study. The duration of each experiment was between 1.5 and 2 seconds, with 3 seconds between experiments. Due to the nature of the clock synchronization algorithm, the uncertainty intervals rise as the studies progress, but the maximum value of uncertainty is the same for all studies irrespective of the number of experiments in them. This follows from the proof of bounded uncertainity intervals in Section 4.1, which shows that the maximum value is dependent on only the network latency.

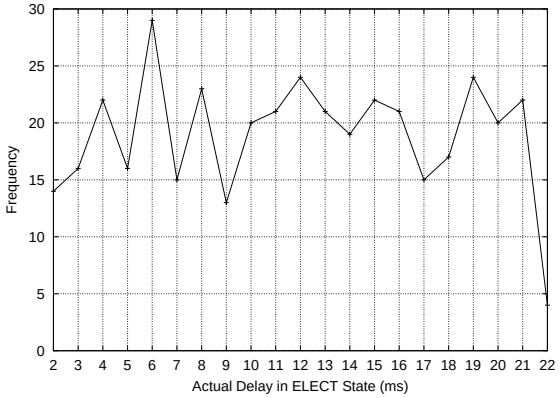
Figures 5(b) and 5(c) demonstrate the minimum holding time properties of Loki. To get those plots, we ran a set of 400 runs of the election protocol. In different runs, we varied the amount of time the three nodes spent in the `ELECT` state by artificially inducing a delay inside that state. The delay induced was between 1 and 20 ms in increments of 1 ms (a constant induced delay per study). Then we tried to inject the fault `nelectfault` into node *nsx*. `nelectfault` is triggered when all three nodes are in the `ELECT` state. Its definition is given in Table 1. Figure 5(b) shows the number of experiments for each value of actual time spent by all three nodes in the `ELECT` state. As seen, this distribution of the actual time spent is not uniform, even though the induced delays were. We believe this is due to OS scheduler artifacts. Figure 5(c) shows a plot of the fraction of faults successfully injected (injection efficiency) versus the actual time spent by all nodes in the `ELECT` state rounded to the nearest millisecond. It can be seen that the injection efficiency is nearly 100% for all state-holding times above 5 ms. For lower state-holding times, the 10 ms OS scheduler timeslice causes the injection efficiency to drop. However, the efficiency is still greater than 60% for all holding times above 1 ms.

6.2.2 Intrusiveness

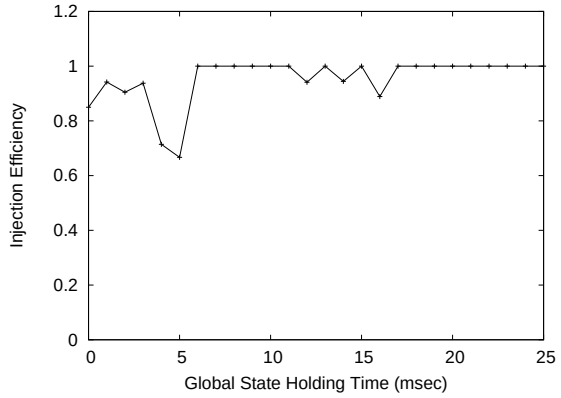
Intrusiveness is another important metric for any instrumentation tool. It is important to minimize intrusiveness as much as possible in order to get the most accurate measurements possible. Since



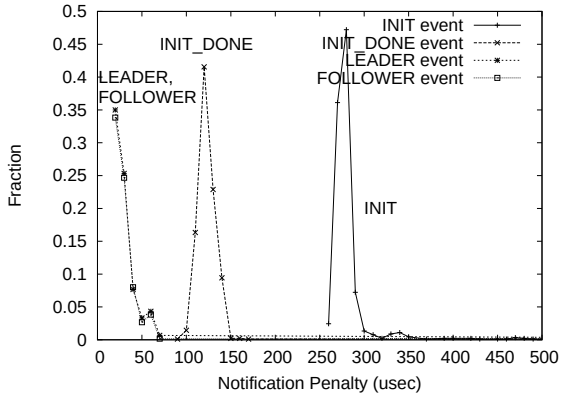
(a) Uncertainty Intervals over the Course of a Study



(b) Actual Delay Distribution



(c) Injection Efficiency



(d) Notification Penalty

Figure 5: Loki Performance Results

the Loki runtime attaches itself to the application being instrumented, it might be argued that Loki is more intrusive to the application than a tool that monitors the application while running as a separate process. However, we claim that this is not so. Even if an instrumentation tool runs as a separate process, the OS would have to schedule its threads. Those threads would compete for the same CPU resources that Loki threads now compete for; hence, the situation would not be any different. The only other ways that the application interacts with Loki are through the `notifyEvent` and `injectFault` methods. The `injectFault` method is invoked by the fault parser and runs in a different thread from the application, and hence does not contribute to intrusiveness beyond the competition for CPU resources mentioned above. However, the `notifyEvent` call is made by

the application thread and does affect the intrusiveness. Hence, for purposes of the current Loki implementation, we characterize intrusiveness by the amount of time it takes a `notifyEvent` call to return. We call this time the *notification penalty*.

We measured the distribution of the notification penalty for node *nsx* during the execution of the election protocol application, as described in the previous section. Figure 5(d) shows the distribution for four different event notifications in the election protocol. It can be seen that these distributions are fairly narrow in width, but tend to have long tails. However, the peaks are different for different events. This is explained as follows. The `INIT` event notification initializes the state machine and is the first event notification that the node sends to indicate that it is functional. This `INIT` event notification also informs all other components of the runtime about the new state machine that has come up. Therefore, it is seen to take more time than other events. The `LEADER` and `FOLLOWER` events do not need any remote notifications. These state transitions are completely local, requiring no IPC, and represent the best case for normal notifications. The `INIT_DONE` event requires that notifications be sent to the other two state machines. Its notification therefore requires more time than those of `LEADER` and `FOLLOWER` events, since IPC is involved. More importantly, each of the four notifications takes between 30 μs in the best case to approximately 280 μs in the worst case. Hence, it can be argued that the notification penalty, and thus intrusiveness due to Loki, are reasonably low.

7 Limitations of Loki

The Loki fault injector, as it is currently implemented, has some limitations. First, Loki has been implemented in C/C++ on Linux on an x86 architecture, and the Loki library has to be linked into the application code. Availability of application sources makes this easier. However, instrumentation of applications that have been written in languages other than C++, or for which source code is unavailable is possible through use of a C++ wrapper, as we did for the Ensemble case study [14].

Second, Loki has been designed for injecting faults at the granularity of hundreds of microseconds. In its current form, it cannot inject faults at a finer granularity. Third, the linear clock drift assumption made by Loki may not hold if experiments are very long (running into months), or if

there are significant changes in temperature during the experiment. Finally, Loki does not currently have a pre-built library of probes for different fault models. A user must therefore build such probes himself/herself.

8 Related Work

Many of the earlier fault injectors were developed with specific systems and fault types in mind. Fault injection and evaluation tools for stand-alone systems include MESSALINE [3], FIAT [20], FERRARI [15], and FTAPE [24]. The fault injection techniques used in those tools encompass hardware-implemented fault injection (e.g., MESSALINE) and software-implemented fault injection (e.g., FIAT). Though it would be non-trivial to extend those tools to evaluate other systems (particularly distributed systems), they served their intended purposes well.

Past research work in monitoring and measurement of distributed systems is also interesting, and is related to the work in this paper. JEWEL [16] is a measurement system that performs online monitoring, evaluation, and visualization, as well as offline analysis, of distributed systems. SPI [4] provides a flexible framework for distributed system evaluation and visualization, and is based on the event-action programming model, in which "ea-machines" observe events in the system and execute actions. Although both of those tools can be extended for fault injection, they do not have the required flexibility, and cannot ensure that faults have been injected in the correct global states.

Several of the existing fault injectors have been targeted specifically towards distributed systems, and are well-suited for their intended applications. However, they do not have all the requirements necessary for global-state-based distributed system fault injection. In CESIUM [1], the distributed execution of the processes in the system is simulated in a single address space, and fault injection is performed within the simulation. Though this approach gives the user a great deal of control over his/her distributed system, it cannot fully mirror the operation of the system in a real environment. DOCTOR [12] injects memory, CPU, or communication faults probabilistically or based on past history, and has an integrated workload generator. EFA [10] was designed for system verification, and generates random fault cases, user-defined fault cases, and/or fault cases derived from an analysis of

the source program of the system. It allows the user to express fault locations and types using a special language, and provides support for controlling the sequence of concurrent events in a distributed system. The Orchestra fault injector [9] was developed for fault injection in protocol stacks, and integrates into the system under study as a layer in the protocol stack. NFTAPE [22] was designed with flexibility and portability in mind, and 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 and are similar in concept to the probes in Loki. NFTAPE uses a modular triggering mechanism and supports a variety of triggers, such as time-based triggers and event-based triggers. NFTAPE has been used to inject various types of faults into systems, including hardware-based, software-based, and simulation-based faults.

All of the tools described above have been successfully applied to many systems. However, to the best of our knowledge, none of the earlier fault injectors have all the capabilities desired for global-state-based fault injection, namely flexibility in fault types, fault injection based on global state, verification that the injections were correct, and accurate computation of a wide range of user-specified measures. Loki has been designed with those capabilities in mind, and we believe that this makes Loki a unique tool for global-state-based fault injection in distributed systems.

9 Conclusions

The Loki fault injection tool has tackled the challenging problem of global-state-based distributed system fault injection by tying together fault injection based on a partial view of the global state, optimistic synchronization, and offline analysis. A flexible measure language, statistical estimation of user-specified performance and dependability measures, and an easy-to-use user interface have also been designed and developed as part of the fault injection tool.

Flexibility, in regards to the choice of fault types and the type of system under study, has been one of the main philosophies of Loki. The experimental results presented in this paper indicate that Loki's fault injection efficiency is high. We believe that the low intrusion and high efficiency of global-

state-based fault injection in Loki, combined with its flexibility and statistical measure estimation capabilities, will make it an invaluable tool in distributed system evaluation. The usefulness of the Loki fault injector has been illustrated in two case studies involving the evaluation of real-life, mature distributed systems.

References

[1] G. Alvarez and F. Cristian. Centralized failure injection for distributed, fault-tolerant protocol testing. In *Proc. of the 17th IEEE Intl. Conf. on Dist. Comp. Systems (ICDCS'97)*, pages 78–85, May 1997.

[2] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martin, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. on Software Engg.*, 16(2):166–182, February 1990.

[3] J. Arlat, Y. Crouzet, and J. C. Laprie. Fault injection for dependability validation of fault-tolerant computer systems. In *Proc. of the 19th Intl. Symp. on Fault-Tolerant Comp. (FTCS-19)*, pages 348–355, June 1989.

[4] D. Bhatt, R. Jha, T. Steeves, R. Bhatt, and D. Wills. SPI: An instrumentation development environment for parallel/distributed systems. In *Proc. of the 9th Intl. Parallel Processing Symp.*, pages 494–501, 1995.

[5] W. G. Bouricius, W. C. Carter, D. C. Jessep, P. R. Schneider, and A. B. Wadia. Reliability modeling for fault-tolerant computers. *IEEE Trans. on Computers*, 20(11):1306–1311, 1971.

[6] R. Chandra, M. Cukier, R. M. Lefever, and W. H. Sanders. Dynamic node management and measure estimation in a state-driven fault injector. In *Proc. of the 19th IEEE Symp. on Reliable Dist. Systems*, pages 248–257, October 2000.

[7] K. Chandy and L. Lamport. Distributed snapshots: Determining the global states of distributed systems. *ACM Trans. on Computer Systems*, 3(1):63–75, 1985.

[8] K. Chandy and J. Misra. An example of stepwise refi nement of distributed programs: Quiescence detection. *ACM Trans. on Prog. Languages and Systems*, 8(3):326–343, July 1986.

[9] S. Dawson, F. Jahanian, T. Mitton, and T. L. Tung. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Proc. of the 26th Intl. Symp. on Fault-Tolerant Comp. (FTCS-26)*, pages 404–414, June 1996.

[10] K. Echtle and M. Leu. The EFA fault injector for fault-tolerant distributed system testing. In *Proc. of the IEEE Workshop on Fault-Tolerant Parallel and Dist. Systems*, pages 28–35, 1992.

[11] C. E. Ellingston and R. J. Kulpinski. Dissemination of system time. *IEEE Trans. on Communications*, COM-21:605–623, May 1973.

[12] S. Han, K. G. Shin, and H. A. Rosenberg. DOCTOR: An integrated software fault injection environment for distributed real-time systems. In *Proc. of the Intl. Computer Performance and Dependability Symp.*, pages 204–213, 1995.

[13] 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.

[14] K. R. Joshi, M. Cukier, and W. H. Sanders. Experimental evaluation of the unavailability induced by a group membership protocol. In *Proc. of the 4th European Dependable Comp. Conf. (EDCC-4)*, pages 23–25, October 2002.

[15] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A tool for the validation of system dependability properties. In *Proc. of the 22nd Intl. Symp. on Fault-Tolerant Comp. (FTCS-22)*, pages 336–344, July 1992.

[16] F. Lange, R. Kroeger, and M. Gergeleit. JEWEL: Design and implementation of a distributed measurement system. *IEEE Trans. on Parallel and Dist. Systems*, 3:657–671, November 1992.

[17] R. M. Lefever, M. Cukier, and W. H. Sanders. An experimental evaluation of correlated network partitions in the Coda distributed fi le system. In *Proc. of the 22nd IEEE Symp. on Reliable Dist. Systems*, October 2003.

[18] K. Marzullo and G. Neiger. Detection of global state predicates. In *Proc. of the Fifth Intl. Workshop on Dist. Algorithms*, pages 254–272, 1991.

[19] D. Powell, E. Martins, J. Arlat, and Y. Crouzet. Estimators for fault tolerance coverage evaluation. In *Proc. of the 23rd Intl. Symp. on Fault-Tolerant Comp. (FTCS-23)*, pages 228–237, 1993.

[20] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson, and T. Lin. FIAT – Fault injection based automated testing environment. In *Proc. of the 18th Intl. Symp. on Fault-Tolerant Comp. (FTCS-18)*, pages 102–107, 1988.

[21] K. G. Shin and Y. H. Lee. Measurement and application of fault latency. *IEEE Trans. on Computers*, C-35(4):370–375, April 1986.

[22] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer. NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proc. of the 4th IEEE Intl. Computer Performance and Dependability Symp. (IPDS-2K)*, pages 91–100, March 2000.

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

[24] T. Tsai and R. Iyer. Measuring fault tolerance with the FTAPE fault injection tool. In *Proc. of the Eighth Intl. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 26–40, September 1995.

A Operational Details

This appendix contains many operational details of Loki that are supplemental to the rest of this paper. These details include the operation of the Loki runtime and an explanation of how Loki is used to evaluate a system.

A.1 Operation of the Loki Runtime

This section describes in detail how the Loki components fit together and how they coordinate to achieve the desired functionality of the runtime. We describe normal operation, operation during the crash and restart of nodes during an experiment, and experiment completion.

A.1.1 Normal Operation

At the beginning of an experiment, the central daemon starts one local daemon per host. The local daemons connect to each other and to the central daemon using TCP. The central daemon then instructs the local daemons to start the nodes that the user specified should be started at the beginning of an experiment. After the initial nodes are started, new nodes can enter the system or existing nodes can leave the system at any time during the experiment.

When a node starts up, its state machine transport sends an IPC connection request to its local daemon. The daemon creates a new IPC channel for communication with the node, and spawns a new thread to service the new channel. To send a notification message to a set of remote state machines, a state machine transport forwards it through its local daemon. To do so, it sends its local daemon the state change notification along with the list of state machines to be notified. The daemon looks up the local daemons of each of the recipient state machines and forwards the notification to them using TCP. These daemons in turn forward the notification to the state machine transports of the recipient state machines using IPC. If there is a notification for a state machine that is currently not executing, the notification is discarded with a warning message in the experiment log[2]. Note that the local daemon of the sending state machine needs to send only one notification to the local daemon on

[2]The log is different from the local timelines and is useful for debugging.

a remote host, even if multiple state machines on the host are receiving it. Also, notifications between state machines on the same host do not require the TCP step, and hence have a lower latency, thus increasing Loki's precision.

At every change in the partial view of global state of a node, its parser checks all the fault triggers and instructs the probe to inject the required faults into the node. Also, the node's recorder records every local state change and fault injection, along with their occurrence times, to the local timeline. When the node exits normally, its local daemon and all the other state machines are notified of its exit.

A.1.2 Node Crash and Restart

When a node crashes, the Loki runtime detects the crash in one of two ways. First, the IPC channel between the node and its local daemon is deleted and the local daemon detects this event[3]. Second, the local daemon functions as a *watchdog* and monitors all its nodes for heartbeats. If one of the nodes times out, the local daemon considers it crashed. The user is allowed to choose the timeout value suitable for his/her application. After a node crash is detected in either way, the local daemon writes a crash event to the local timeline of the node, and notifies all the other local daemons of the crash.

When a node crashes, the distributed application under study could restart it, possibly on a different host. To distinguish between a new node and a restarted node, the node's state machine checks its local timeline as soon as it starts up to see whether it is new or restarted. The local timeline is stored on a distributed file system to facilitate restarting on a different host. If a node has been restarted, its state machine writes a restart event to the local timeline. Its state machine transport then connects to the local daemon, and the local daemon notifies all the other local daemons that the node has restarted. The restarted node obtains state updates from all the other nodes to reconstruct its partial view of global state. After that, the node executes like a normal one.

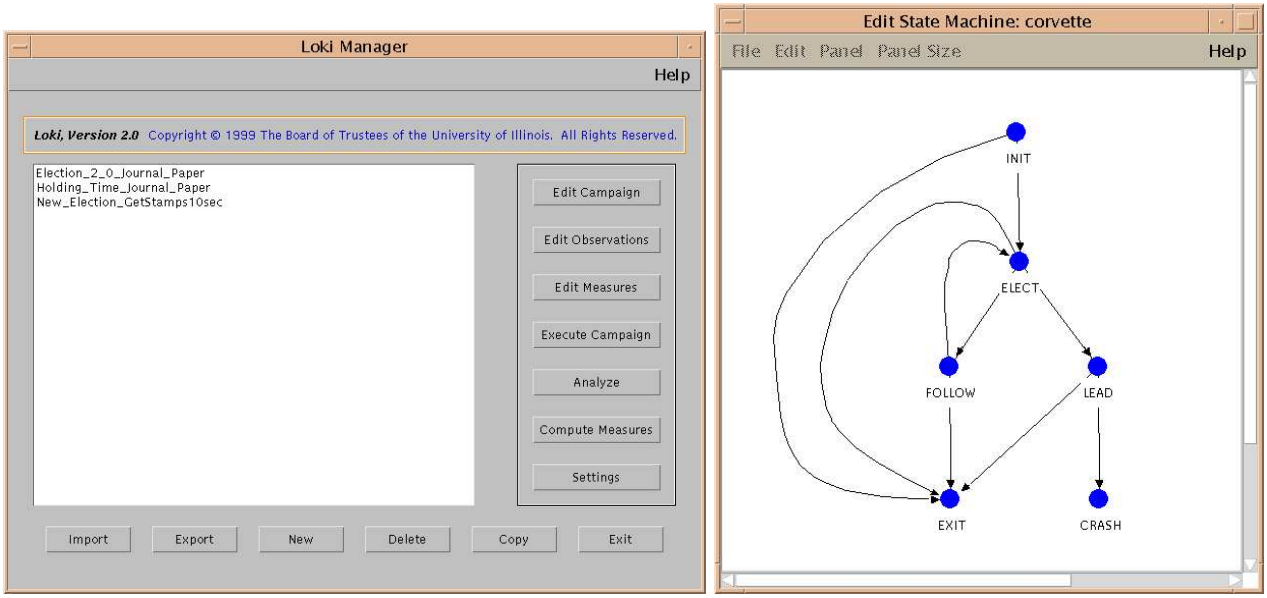
[3]Note that since the deletion is performed by the signal handler for the node, the user should be careful if he/she has overridden the signal handler in his/her application.

A.1.3 Experiment Completion

The problem of detection of experiment completion relates to the problem of *distributed quiescence detection* (or distributed termination detection) [8]. In general, it is impossible for the Loki runtime to automatically detect experiment completion (without any help from the application), because it might be that not all of the components of the application are instrumented with the runtime. Thus, even if all the local daemons simultaneously observe that there are no nodes of the application running and hence decide that the experiment is complete, there might be other uninstrumented components running that could restart some instrumented components after the daemons' decision. Such situations can be avoided if all the application components whose execution is considered necessary for the experiment (including those that restart other application components) are instrumented with the Loki runtime. That would prevent the daemons from incorrectly concluding that the experiment is complete when crucial application components are in fact still running, and also prevent "restart messages" in transit from causing application components to restart after the daemons have determined that the experiment has completed.

If the system is so instrumented, then the detection of experiment completion can be done as follows. When a local daemon observes that all the nodes in its host have either crashed or exited, it sends an "experiment complete" message to the central daemon; the central daemon deems the experiment complete if all the local daemons have reported that the experiment is complete. The current Loki runtime implements this mechanism to detect experiment completion.

At the end of an experiment, the central daemon shuts down all the local daemons and starts the next experiment. Loki allows the user to specify timeout values for experiments, to avoid problems when a distributed application fails to terminate itself. Thus, the central daemon keeps a timer on the length of an experiment. If a timeout value is reached, the central daemon instructs the local daemons to kill the nodes, and concludes the experiment. Of course, as is the case for any timeout-based fault detection mechanism, the user has to choose a timeout value reasonable for his/her particular application.



(a) Loki Manager (b) State Machine Editor

Figure 6: Some Screen Shots from Loki's Graphical User Interface

A.2 Using Loki to Evaluate a System

A user must take several steps in order to evaluate a system using Loki. They include campaign specification, system instrumentation, measure specification, campaign execution, and offline processing. An extensive front end was developed for Loki to guide the user through the specification and campaign evaluation processes involved in Loki's fault injection process. The main window of the front end is shown in Figure 6(a). It contains a button for each step that must be performed while conducting a fault injection campaign using Loki. In addition, during the execution of campaigns, the front end provides the central daemon functionality described in Section 3.

A.2.1 Campaign Specification

The first step in the specification of a campaign is to name it and divide it into studies as discussed in Section 2. After giving a study a name, the user defines it by setting several parameters, registering local daemons, and dividing his/her system into nodes. The parameters include the number of experiments to be conducted for the study, the time to wait between experiments, an application timeout

value, and a port number that local daemons use to contact the central coordinating daemon. After setting the parameters, the user registers local daemons for each host that might run an instrumented node from the system under study. Registration simply requires a host name, the host's processor clock speed (to convert timelines from clock ticks to μs), and a port number that the daemon should use for communication.

The user must also divide his/her system into nodes. Each node is associated with some parameters, a state machine specification, and a fault specification. The parameters include a host name, application name, application arguments, and heartbeat timeout value. The host name indicates a host on which Loki should launch the node at the start of an experiment. The host name is omitted if Loki is not responsible for starting the node. The application name identifies the node's executable that is instrumented with the Loki runtime. The application arguments that the user gives are passed to the node when it starts. Lastly, the heartbeat timeout is used by a local daemon to determine if the node is alive.

The front end's State Machine Editor, shown in Figure 6(b), allows the user to define a node's state abstraction, which is then translated into a specification that the Loki runtime uses when tracking the partial view of global state required by the study. On the State Machine Editor's canvas, states are represented by vertices. Each state is named, and a list of state machines (to be notified when the state machine enters the state) are selected. The list helps Loki maintain the partial view of global state. States are connected with directed connection lines identifying event-triggered transitions. These transitions also require the user to identify the event causing the transition. The counterpart to the State Machine Editor is a Fault-Trigger Specification Editor that registers all the faults that can be injected in the node. Each fault is registered with a name, a fault trigger, and an indication of whether the fault should be injected "once" or "always." As mentioned in Section 2, the fault trigger is a Boolean expression pertaining to some partial view of the system's global state into which the fault should be injected. The final parameter indicates whether the fault should be injected only the first time the fault trigger transitions from false to true, or every time the trigger transitions.

A.2.2 System Instrumentation

Once the campaign is fully specified, the system under study is ready to be instrumented. Four major steps, described below, are required in order to instrument a system for fault injection by Loki. Recall that nodes are instrumented components of the system under study, into which faults may be injected and/or from which state must be tracked.

1. *Implementation of Faults:* The first step is to implement the faults to be injected. For nodes that require the injection of faults, the `injectFault()` method of the node's runtime probe must be coded with an implementation of those faults. The only input parameter to the `injectFault()` method is a fault name from those specified in the node's fault trigger specification. The method returns the time at which the fault is injected. This technique of having the user implement the desired faults provides a high degree of flexibility by separating the user's policy from the overall mechanism of fault injection. Implementation of a non-trivial probe currently requires significant effort. In the future, we plan to ease the process of implementing faults by providing a library of Loki probes for commonly used faults.

2. *Event Notification:* The second step is to indicate the occurrence of events that pertain to transitions in the state machine of a node. This step involves making a call to the probe's `notifyEvent()` method at appropriate places in the system under study to indicate when events occur. The parameters to `notifyEvent()` are the name of the event and the time at which it occurred. Event notification needs to be done only from those nodes whose state must be tracked.

3. *appMain:* The next step is to use an `appMain()` method in place of the standard `main()` method so the node's Loki runtime can initialize before calling `appMain()`.

4. *Node Invocation:* The final step is to modify the invocation commands for nodes. This change involves the arguments passed to the nodes. In order for the Loki runtime to identify study parameters and locate the appropriate local daemon, the arguments to a node must be its corresponding *StudyFile* and the study's *DaemonContactFile*. Both of these files are automatically

created during the specification of a fault injection campaign by the Loki front end. This modification is never an issue if the system's nodes are not dynamically started during an experiment.

The current implementation of the Loki runtime is in C++. For that reason, instrumented nodes must be either written in C/C++ or interfaced with C++.

A.2.3 Measures Specification

In addition to the campaign specification needed to perform experiments, the user needs to provide a measures specification to compute desired measures for the campaign. Again, the measures specification is managed from the Loki Manager and is composed of both study-level and campaign-level measure specifications. A user creates a study-level measure by giving it a name and defining its (subset selection, predicate, observation function) triples as described in Section 5.2. After defining a study-level measure, Loki compiles it into an executable that computes the values that make up its sample from the global timelines of the selected experiments. Similarly, each campaign-level measure is named, defined, and compiled into an executable. In the definition, the user assigns one of the three campaign-level measure types described in Section 5.3. If the measure is simple sampling, the user selects study-level measures whose final observation values will be used in the campaign-level measure's sample. For stratified weighted measures, the user selects study-level measures and assigns them weights. Finally, for stratified user measures, the user defines a function that combines the means of study-level measures. The executable that Loki compiles for a campaign-level measure takes study-level measure samples as input, and generates appropriate statistical properties as output.

A.2.4 Campaign Execution

Once the user has specified a campaign and its measures, and instrumented the system under study, the campaign is ready to be executed. Campaign execution is managed from an Experiment Manager that is launched from the Loki Manager. The user assigns six execution parameters, which pertain to timestamps for use in clock synchronization, as described in Section 4.1. First, the user is asked to indicate how many synchronization messages should be passed before and after either studies or

experiments. The user also specifies the amount of time that should elapse between synchronization messages, for both *before* and *after* messages. The next parameter determines whether *before* and *after* should refer to before and after each study or before and after each experiment. The final parameter is a port number on which the synchronization messages should be passed. Following parameter assignment, the user starts automated campaign execution from the Experiment Manager.

Behind the scenes, the Experiment Manager oversees clock synchronization message passing and performs experiment coordination, daemon management, and experiment monitoring. The pseudocode in Algorithm 1 demonstrates the general algorithm that the Experiment Manager follows. The `CampaignExecution()` algorithm runs in its own thread, which is referred to by the same name. Below, *lokid* is used to refer to local daemons.

Algorithm 1: Pseudocode for campaign execution in the Experiment Manager.

```
CAMPAIGNEXECUTION()
foreach _study_
    /* Start local daemons phase */
    foreach _host_
        EXEC(lokid)
        ACCEPT(remote_lokid_connection)

    /* Experiment execution phase */
    foreach _experiment_
        if (collect_timestamps_after_every_experiment
        = true ) or (_experiment_ = 0)
            foreach _host_
                EXEC(getstamps)
        foreach _lokid_
            SEND(start_experiment_msg, _lokid_)
        foreach _node_
            if REQUIRESRUNTIMESTARTUP() =
            true
                SEND(start_node_msg, GETLOKID(
                _node_))
        while not (experiment_complete) and
        (TIME() < app_timeout)
            PROCESSDAEMONMSGS(daemon_msg_buffer)
        if TIME() > app_timeout
            foreach _lokid_
                SEND(timeout_msg, _lokid_)
        foreach _lokid_
            RECEIVE(experiment_complete_msg,
            _lokid_)
        CHECKFOREXPERIMENTERRORS()

    /* Study completion phase */
    CHECKFORSTUDYERRORS()
    foreach _lokid_
        SEND(kill_msg, _lokid_)
    foreach _host_
        EXEC(getstamps)
```

Three aspects of the algorithm's implementation that deserve discussion are remote process invocation, communication between the central daemon and local daemons, and experiment and study error detection. The Experiment Manager makes extensive use of remote process invocation for executables such as `getstamps` and `lokid`. The `ssh` secure shell client is used as a remote execution mechanism for starting processes on remote hosts. The `stdout` and `stderr` I/O files of remote

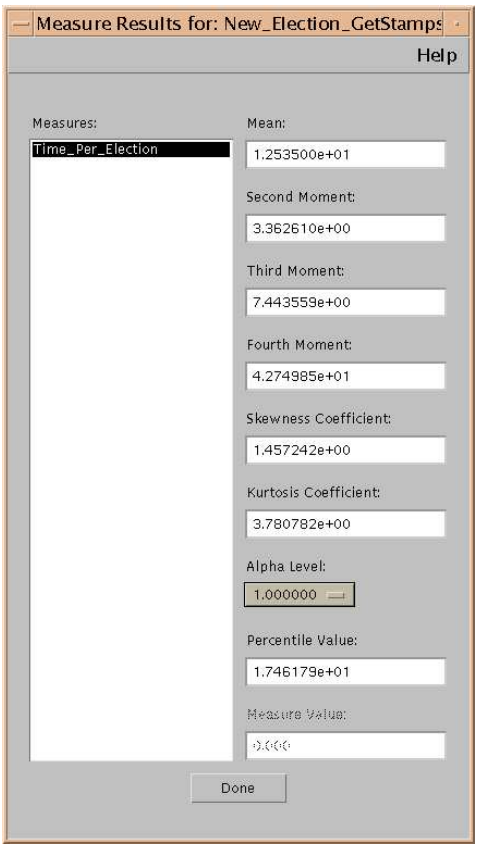


Figure 7: Measure Results Window

processes are redirected to appropriate log files to be examined by the front end. Although `stdout` and `stderr` I/O files are useful for logging purposes, they are not adequate for all communication required by the front end. In particular, TCP sockets are needed for communication between the central daemon and the local daemons. On the central daemon side (the Loki front end), each socket connection is maintained in a separate thread, called a `SocketThread`. When messages arrive from the local daemons, they are tagged with identifying information and queued in a daemon message buffer. The message buffer has a lock for concurrent access. The `CampaignExecution` thread handles the messages as designated in Algorithm 1. Each `SocketThread` also maintains a socket lock for an outgoing message buffer. Throughout the campaign execution process, the Experiment Manager must monitor for errors. It does so by examining the log files created by remote processes, the messages received on sockets, and the status of socket connections. It handles errors by archiving failed experiments for later examination.

A.2.5 Offline Processing

After the user has finished campaign execution, the campaign is ready to be processed by Loki. As described earlier, there are two phases to the offline processing: experiment analysis and measure computation. Each step is invoked from the Loki Manager. Experiment analysis performs the analysis phase of Loki, as described in Section 4, to create global timelines and determine the correctness of fault injections from the raw data collected during the campaign's execution. Following analysis, the user can examine details such as the α and β values for the clock synchronization, the timing information on when particular states were entered, the status (correct injection, incorrect injection, or not injected) and timing information for faults, and the global timeline for the experiment. After the analysis is performed, measure computation (as described in Section 5) is carried out from the Loki Manager. Once that has been done, study-level and campaign-level measure executables are run in order to calculate statistical properties of the desired campaign measures. These properties are then displayed to the user in the front end's Measure Result Window, seen in Figure 7. The properties include the first four central moments, the skewness coefficient, the kurtosis coefficient, and approximate percentile values.