Fault Injection Based on a Partial View of the Global State of a Distributed System[1]

Michel Cukier, Ramesh Chandra, David Henke, Jessica Pistole, 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
{cukier, ramesh, henke, pistole, whs}@crhc.uiuc.edu

Abstract

Validating distributed systems is particularly difficult, since failures may occur due to a correlated occurrence of faults in different parts of the system. This paper describes the basis for and preliminary implementation of a new fault injector, called Loki, developed specifically for distributed systems. Loki addresses issues related to injecting correlated faults in distributed systems. In Loki, fault injection is performed based on a partial view of the global state of an application. In particular, facilities are provided to pass user-specified state information between nodes to provide a partial view of the global state in order to try to inject complex faults successfully. A post-runtime analysis, done using an off-line clock synchronization and a bounding technique, is used to place events and injections on a single global timeline and determine whether the intended faults were properly injected. Finally, observations containing successful fault injections are used to estimate specified dependability measures. In addition to describing the details of our new approach, we present experimental results obtained from a preliminary implementation in order to illustrate Loki's ability to inject complex faults predictably.

1. Introduction

Distributed systems are increasingly prevalent, and are being used more and more in applications that must be dependable. When a distributed system must be highly available or reliable (e.g., for medical or monetary reasons), it is necessary to validate it to demonstrate that it meets a particular availability or reliability specification. Unfortunately, distributed systems have characteristics that make it more difficult to validate them than other computing systems. In particular, the behavior of such systems often depends on subtle event and state-change orderings between different processes or objects in the system. Furthermore, the effect that faults have on such a system can depend critically on the precise global state that the system is in when they occur. For example, in the case of correlated faults, the phase of recovery the system is in when the second fault occurs may radically change its effect on the system. The faults that lead to system failure can thus be very complex and have effects that depend on the global state of the distributed system when they occur.

Knowledge of the global state of a distributed system during fault injection is thus desirable, but usually impossible to obtain in practice. In particular, attempts to control the execution of a distributed application or system to obtain global state information are inherently intrusive, and hence may affect the outcome of an experiment in unacceptable ways. Furthermore, attempts to communicate state information between nodes of a fault injector in a non-intrusive manner may fail, since if the application or system is not controlled, the information is inherently out of date by the time it is communicated to a remote node. Therefore, if one accepts the premise that a fault injector must be as non-intrusive as possible, one must accept the possibility that the fault will not be injected where intended, and will have different effects than intended. This has serious consequences for both of the common purposes of fault injection: fault removal and dependability assessment. In the case of fault removal, if a fault causes a system failure, the state that the system was in when the fault was injected may not be known, making it difficult to remove the fault. In the case of dependability assessment, the injection may occur in an unintended and unrepresentative state, biasing the computed dependability measures. Building a fault injector for distributed systems is thus a difficult task.

There are many fault injection and measurement tools already in existence, several of which are targeted specifically to distributed systems. Many have made important steps toward addressing the problem outlined in the previous paragraph. EFA is one such injector [Echtle 92]. EFA allows for the exchange of information between local nodes, but it does not provide a formal means of specification of this communication, making fault injection experiment design difficult and verification of the proper injection of faults in a certain state complex. Orchestra [Dawson

[1] This research has been supported by DARPA Contract F30602-96-C-0315.

96] is a flexible distributed system fault injector that can inject faults anywhere in a layered protocol stack. Orchestra allows for fault injection based on the local state of a node, but does not provide a formal means for exchanging information between nodes. SPI [Bhatt 95] provides a flexible framework for distributed system evaluation and visualization. Based on the state of the system, measurements can be taken globally and actions can be performed. SPI was developed primarily with measurement and evaluation in mind, and was not designed specifically for fault injection; however, SPI could be used for fault injection. NFTAPE [Stott 99] permits the injection of CPU-, memory-, I/O- and communication-related faults in a networked system. A design goal of NFTAPE is to support the injection of a wide variety of fault types and be extensible, in the sense that new fault types can by added by writing a small block of code to perform an injection. DOCTOR [Han 95] is a fault injection environment for distributed real-time systems that supports the injection of memory, CPU, and communication faults. Finally, CESIUM [Alvarez 95] takes a centralized simulation-based approach to fault injection. The distributed execution of the processes in the system under study is simulated on a single machine in a single address space, with network interaction and system clocks simulated by the CESIUM environment. These fault injectors are well-suited to the applications for which they were intended. However, they do not support injections based on global state.

This paper specifically addresses that issue, by developing the necessary theory and algorithms to make use of global state in a distributed system fault injector, and providing results from a prototype that show that our ideas are sound. More specifically, we 1) develop a method to specify the faults that should be injected in a distributed system, where the injection can depend on the global state of the system, 2) develop a method for tracking the global state in a distributed system at the level of detail necessary to perform injections, and 3) develop a method to determine whether a given injection of a fault occurred as intended, using results from an off-line clock synchronization and bounds analysis. In addition, we have implemented these results in a prototype fault injector for distributed systems, called Loki, and provide experimental results that show that our approach is indeed sound, and that the implementation can efficiently and predictably inject faults based on the global state of an application.

The remainder of this paper is organized as follows. Section 2 presents an overview of the Loki architecture and describes the steps taken when conducting a fault injection campaign with Loki. Section 3 describes the theory and algorithms that support these steps, including the Loki runtime and clock synchronization protocols. Section 4 presents results from the fault injection of a simple leader election protocol, illustrating Loki's ability to inject faults correctly based on a partial view of the global state of an application. Finally, Section 5 concludes the paper and presents research directions that we are currently pursuing.

2. Loki Overview

This section presents an overview of Loki. More details can be found be found in [Henke 98, Pistole 98], which first introduced the concepts we describe here. Before describing in detail the steps Loki uses to perform a fault-injection campaign, it is useful to introduce the notion of a partial view of the global state, and the architecture of Loki itself.

The first, and most basic, concept used by Loki is the notion of state. In most distributed systems, each component only knows the state of the other components in the system at the time of synchronized events. At other times, knowledge of the state of any given component by other components is not guaranteed. In order to evaluate and remove faults in a fault-tolerant distributed system, it may be necessary to inject faults in a component based on the state of one or more other components, even at a time when the complete state of the system is unknown. To inject such faults, it is not necessary to know the global state of the system at all times; instead, a "partial view" of the global state of the system is sufficient. Informally, we mean the state of "interesting" components and knowledge of "interesting" state changes. The portion of state that is interesting depends on the particular system being studied and the faults one desires to inject.

With this in mind, a distributed system (the "system under study") can be subdivided into the pieces that are necessary for a given fault injection campaign. These pieces, which include both the portions of the distributed system into which faults are to be injected and the parts from which state information must be obtained, will be called *nodes* in the following discussion. The Loki run-time architecture provides a framework for maintaining a partial view of the global state using a set of nodes, and for conducting fault injections using that partial view of the global state. More specifically, Loki maintains a partial view of the global state by using state machines associated with nodes that communicate with one another. Figure 1 shows the global architecture of a system using Loki. Each node in the system is coupled with a state machine that tracks the state of that node relevant to the fault injection campaign. These state machines send *state change notifications* (messages containing information about the global state of the system) to subsets of all of the state machines in the system as necessary to maintain the partial view of the global state needed for fault injection.

Because Loki is designed to be as non-intrusive to the application as possible, it does not block the system under study while waiting for a notification to arrive at another state machine. Accordingly, a system may actually change state again by the time a state change notification reaches its target state machine(s). This implies that the partial view of the global state seen by Loki is not always correct. The correctness of a particular state is determined by the holding time of the global state and the time needed to transmit a notification after the state is reached. Any fault

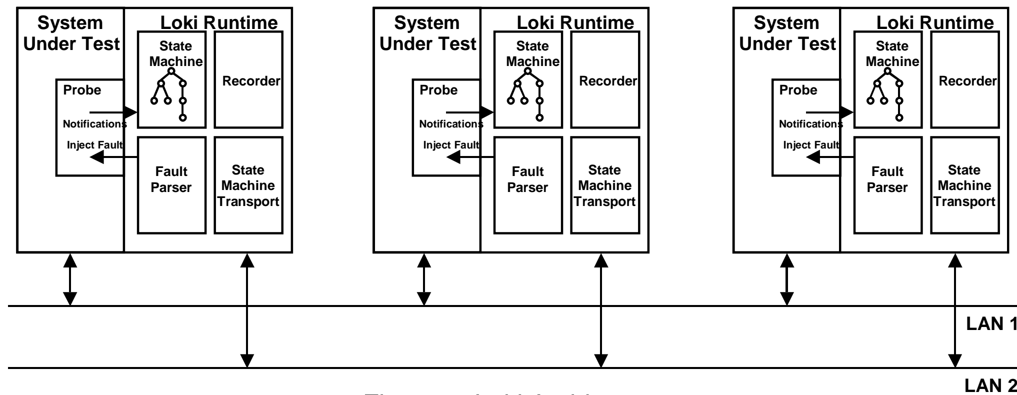

Figure 1. Loki Architecture

injection based on a global state of the system must thus be checked after the experiment is completed, to see whether the fault was injected as intended.

In order to check whether an injection was performed correctly, all events (state changes and fault injections) must be placed on a single global timeline. One way to do this would be to use an on-line clock synchronization to maintain a global clock to timestamp events. Loki does not do this, since an on-line clock synchronization would be expensive and intrusive. Furthermore, on-line clock synchronization is unnecessary, since faults that are based on a “global” time can be triggered by a combination of the partial view of the global state and a local timekeeping mechanism. Determination of the global time is necessary only for checking the correctness of an injection and computing dependability measures, and thus can be done off-line.

Given these basic concepts, we can describe, at a high level, the organization of a fault injection campaign and the components of Loki that are used to carry out the campaign. At the highest level, a campaign can be divided into four phases: 1) an initial message-passing phase, 2) an injection of faults and collection of observations, 3) a second message-passing phase, and 4) the determination of whether each injection was performed correctly, and computation of the desired measures. As will be seen in the next section, the two message-passing phases are used to provide information necessary in order to put events and fault injections on a single timeline. The fault injection and collection of observations are done during a run of the distributed application, using the Loki runtime, as will be described.

The first and third steps are used to place all recorded observations from the fault injection onto a single global timeline. The algorithm for accomplishing this is based on the assumption that the drift of any given machine’s clock with respect to “perfect time” is linear. To build the global timeline, messages are sent between all machines of the system under study, and the corresponding message-sending and -receiving times are recorded. This is done once before the fault injections are run, once after all fault injections are complete, and sometimes between fault injections. Using these time stamps, the offset and linear drift between each machine and the machine chosen as the reference machine are calculated. Given these values, the local times can be translated into times on the global timeline. In addition, the algorithm provides bounds on each observation time representing the physical limits for the time. Since this algorithm does not require that any special times be recorded during the fault injections, it is non-intrusive to the system.

Step two, the injection of faults and collection of observations, is done using the distributed application run together with the Loki runtime. A copy of the Loki code runs in each node and consists of two parts: a “probe” and a “runtime.” *Probes* perform fault injections into a node of the system under study and provide notification to the runtime regarding the state of the node. A user may construct a probe in three ways. First, the user can utilize the basic probe provided by Loki as an interface only, adding any additional injection and/or notification functionality in the code of the system under study. Second, a user can enhance the provided probe with the particular functionality required for a particular fault injection study, using the basic probe as a template. Third, the user can combine these two approaches, placing some of the instrumentation code inside the probe and some in the system under study. Probes inject faults into a node under the control of the “fault parser,” which is part of the runtime. Probes also provide notifications to the runtime state machine, possibly causing it to change state. The Loki runtime provides the mechanism to initiate a fault injection (via the *fault parser*), maintain state information and initiate state notification to other nodes (via the *state machine*), record timing information concerning state changes that occur and faults that are injected (via the *recorder*), and communicate with other nodes’ runtimes (via the *state machine transport*). The choice of a particular set of state machines and probes, and their assignments to nodes, determines the behavior of the fault injector.

The final step (step 4) determines whether the intended fault has been injected correctly, and computes the desired measures, using observations of experiments in which the fault was injected correctly. Recall that since Loki’s notion

of global state is imprecise in order to be less intrusive, it is possible that fault injections may occur in unintended states. Determining whether a particular fault was injected is done by placing all events (fault- and state-change-related) on a single timeline, using the procedure outlined above. Given a global event timeline for each experiment, Loki determines whether the fault injection(s) occurred in the correct state(s). This is done using a Boolean expression, provided by the user, that drives the injection and is used to determine whether the injection was successful. Finally, if the purpose of the campaign is dependability assessment, the observations of experiments in which faults were injected correctly are used to estimate the dependability of the reliable distributed system. These measures can be quite general, and include all measures based on the sequence of events that occur and times spent in particular states.

3. Activities Necessary to Support a Fault-Injection Campaign

The steps that are required to carry out a fault injection campaign were outlined in the previous section. In this section, we focus on describing the algorithms and mechanisms that are needed to carry out these steps, namely, A) the generation of sets of local timelines of events, B) the transformation of each set of local event timelines into a global event timeline, C) the determination of whether each experiment's fault was injected correctly, and D) the computation of the desired measures. Activity B requires an off-line clock synchronization, as mentioned previously. In the following, we will refer to the injection of a particular, possibly compound, fault during a run of the application as an *experiment*, a set of (repeated) experiments as a *study*, and a set of studies, in which each study injects a different type of fault, as a *campaign*.

Before describing each of these activities, it is helpful to describe the information that a user must specify before a campaign begins. This information is currently specified as a set of files, but will be input via a graphical/form-based interface in the next version of Loki.

Specifically, a *study file* specifies the study name, the application under study, the number of experiments for each study, the time between experiments, and the list of all other input files for a study. A *node file* specifies the node on which each Loki runtime runs, and the port number that is used to communicate with the state machine contained in the runtime. *State machine files* specify, for each state machine, a list of the states and transitions between states (induced by probe notifications), a specification of which other state machines must be notified when the state machine makes a transition to a state, and a specification of which states and events need to be recorded on the local timeline when the machine executes (by default, all events are recorded).

A *fault specification file* for each study specifies the faults that should be injected in the study. This file contains, for each fault, the fault name and a Boolean expression specifying the state(s) in which each state machine should be when the fault is injected and whether the fault should be injected every time the state machine comes to this state or only the first time. The Boolean expression specifies a fault to inject and later helps in testing for proper injection when the output of an experiment is being processed. Each Boolean expression is specified in a standard fashion using the "&" (AND), "|" (OR) and "~" (NOT) operators. There is one fault specification file for each state machine that contains all faults to be injected on that node. When a fault must be injected in several nodes at the same time, the same fault appears in all of the fault specification files associated with all of the nodes where the fault has to be injected.

Finally, the measures that will be estimated are specified in a *measure specification file* for each study. Given this terminology, we can now describe in detail the activities necessary to support a fault injection campaign.

3.1 Calculation of Clock Offsets and Drift Rates

An off-line clock synchronization algorithm is used to calibrate the clocks on the multiple machines on which the fault injector operates, so that observations collected during an experiment can be placed on a single, global, timeline. In particular, we take one machine's clock as the reference, and estimate (with bounds) the offset and drift rate of other nodes' clocks, relative to the reference clock. These offsets and drift rates can then be used to place all events on a single global timeline.

Many different methods have been proposed to calibrate clocks in distributed systems. These methods can be divided into three areas: hardware, software, and some combination of the two. The choice of a method involves a tradeoff between accuracy, intrusiveness, and portability. Hardware methods, e.g. [Mink 90, Hofmann 88], are highly accurate and often not very intrusive. However, they are generally system-specific. Conversely, software methods, e.g. [Lamport 78, Mattern 89, Cristian 89, Lamport 85], increase the portability, but decrease the accuracy and increase the intrusiveness. Finally, the hardware in the methods combining hardware and software, e.g. [Kopetz 87], reduces the intrusion and improves accuracy compared to the pure software methods. Since portability is an important concern to us, and the accuracy provided by software-based synchronization is sufficient, we use a software-based approach. However, the time is read from the hardware clock of the processor. This approach makes use of message exchanges between nodes to calibrate differences in their clocks. Before explaining how we do this, we review the clock model that we use.

Model of physical clocks: Similar to [Ellingston 73, Jezequel 96], we assume that the value displayed on the clock of processor i, $C_i(t)$, can be modeled as:

$$C_i(t) = \alpha_i + \beta_i t + \delta_i \qquad i = 1,..., p \qquad (1)$$

where t is the physical clock time, α_i is the offset at $t = 0$, β_i is the drift of the physical clock, and δ_i models random perturbations and the "reading error" of the clock C_i. δ_i can thus be modeled by $\delta_i = \gamma_i + e_i$ where e_i is a random variable whose distribution is uniform on $[-g_i/2, g_i/2]$, g_i being the resolution of the physical clock. Like [Jezequel 96] we assume that $|\gamma_i| << |e_i|$.

Then, by eliminating t in equation (1) for processors i and j, we obtain the following expression (similar to the expression in [Jezequel 96]):

$$C_j(t) = \alpha_{i,j} + \beta_{i,j} C_i(t) + \delta_{i,j} \qquad i, j = 1,..., p \qquad (2)$$

where $\alpha_{i,j} = \alpha_j - \beta_{i,j}\alpha_i$, $\beta_{i,j} = \dfrac{\beta_j}{\beta_i}$ and

$\delta_{i,j} = \delta_j - \beta_{i,j}\delta_i$.

This expression expresses the time value displayed on one clock in terms of that displayed on another, and suggests a mechanism for computing $\alpha_{i,j}$ and $\beta_{i,j}$ for all "non-reference" clocks in terms of the reference clock. The way we do this is explained next.

Calculation of bounds on $\alpha_{i,j}$ and $\beta_{i,j}$: Following the approach developed in [Duda 87], we calculate physical bounds for $\alpha_{i,j}$ and $\beta_{i,j}$ based on message exchanges between the different processors. Let S_i^k be the physical clock time when the kth message is sent from processor i and let R_j^k be the physical clock time when the kth message is received by processor j. $\tau_{i,j}$ is a positive random variable representing the delay of the message sent from processor i to j. We thus have $C_j(R_j^k) = C_j(S_i^k + \tau_{i,j})$. By using (1) and (2), we obtain the following expression, similar to [Jezequel 96]:

$C_j(R_j^k) = \alpha_{i,j} + \beta_{i,j} C_i(S_i^k) + \beta_j \tau_{i,j} + \delta_{i,j}$ and

$C_j(S_j^k) = \alpha_{i,j} + \beta_{i,j} C_i(R_i^k) - \beta_j \tau_{j,i} + \delta_{i,j}$.

When $|\delta_{i,j}| << \tau_{i,j}$ and $|\delta_{i,j}| << \tau_{j,i}$, the line $C_j = \alpha_{i,j} + \beta_{i,j} C_i$ is located above the set of points $\{C_i(S_i^k), C_j(R_j^k)\}$ (k = 1, …, n_R, n_R being the number of messages sent by processor i to processor j) and under the set of points $\{C_i(R_i^k), C_j(S_j^k)\}$ (k = 1, …, n_S, n_S being the number of messages received by processor i from processor j). Due to the fact that delays are positive, this observation enables the determination of physical bounds for $\alpha_{i,j}$ and $\beta_{i,j}$. Unlike confidence intervals, these physical bounds ensure that the values of $\alpha_{i,j}$ and $\beta_{i,j}$ will be located inside the interval. The bounds for $\alpha_{i,j}$ and $\beta_{i,j}$ are defined in [Duda 87] by:

$\alpha_{i,j}^-, \beta_{i,j}^+ : \min_{\alpha_{i,j}} \max_{\beta_{i,j}} C_j(R_j^k) \le \alpha_{i,j} + \beta_{i,j} C_i(S_i^k), k = 1,..., n_R$,

and $C_j(S_j^k) \ge \alpha_{i,j} + \beta_{i,j} C_i(R_i^k), k = 1,..., n_S$

$\alpha_{i,j}^+, \beta_{i,j}^- : \max_{\alpha_{i,j}} \min_{\beta_{i,j}} C_j(R_j^k) \le \alpha_{i,j} + \beta_{i,j} C_i(S_i^k), k = 1,..., n_R$,

and $C_j(S_j^k) \ge \alpha_{i,j} + \beta_{i,j} C_i(R_i^k), k = 1,..., n_S$

A practical method for obtaining these bounds (described in [Henke 98]) involves the calculation of convex hulls for a set of points via geometric algorithms similar to those in [Sedgewick 84]. In particular, [Henke 98] describes the method we use for building the upper and lower hulls, and a way to obtain the bounds for $\alpha_{i,j}$ and $\beta_{i,j}$ geometrically.

Implementation: The final issue we need to address is how to determine when to exchange the messages to collect timestamps. Since messages are passed and recorded in a normal experiment, the messages themselves could be used to calculate the bounds. However, it cannot be guaranteed that enough messages will be passed to obtain the desired precision for the global timeline. For example, one machine might get update messages from other machines but never send any of its own. Without messages sent from a machine, it is impossible to calculate the adjustment for its timeline. Also, since additional information about the local timestamps would need to be added to normal messages, the program execution would be perturbed. Therefore, all messages used for timestamp synchronization are sent and received during times when no experiment is under way. Alternatively, messages could be sent either before or after the experiments. Due to the linear nature of the clock error, the error in the calculated estimation would grow as adjusted times grow farther away from the time when the synchronization messages were passed. This method will work for short experiments, but as experiments grow longer, the length of the message-passing phase necessary to reduce the error will become prohibitively long.

A better solution is to pass messages explicitly both before and after the execution of an application [Maillet 95]. In the context of Loki, we thus exchange messages between certain experiments or studies, or at the beginning or end of the campaign. This method allows for global timeline estimation over an arbitrary length of time. There are three factors that influence the accuracy of the offset estimation. First, increasing the length of the message-passing phase increases the accuracy of the results. Second, the more messages that are passed during each message-passing phase, the more accurate the results will be. Finally, the longer the experiment takes, the better the estimation will be. Our experimental experience shows that this approach works well in practice [Henke 98], and yields bounds on the estimates of $\alpha_{i,j}$ and $\beta_{i,j}$ that are acceptably small.

3.2 Generation of Sets of Local Event Timelines

Once an initial round of message exchanges to calculate $\alpha_{i,j}$ and $\beta_{i,j}$ is completed, Loki constructs local event

timelines (state changes and fault injections) for each node. One such timeline is built for each node used in an experiment. These timelines are constructed by executing the distributed system under the control of Loki, using the study file to determine the details necessary to carry out the experiment. In the current implementation, the Loki runtime runs as a pair of threads attached to the application itself (which executes as one or more threads); future implementations may permit the runtime to be in a separate process (less intrusion, but lower accuracy).

When the instrumented application begins execution, several steps are taken to initialize the runtime. In particular, each Loki runtime first reads in the study file. Then, using the reference in the study file, each runtime reads in the node file for the study, sorts the nodes in the file according to their nicknames, and arranges them into an array called the *node table*, which contains the node's view of each other state machine's state. Communication channels are then established between all nodes' state machines, by having each runtime connect to all the nodes that have a lower index in the table, and accept connections from all the nodes that have a higher index. Once all connections are established, each process reads its state machine specification file and builds an internal representation for its state machine consisting of a *state table* and an *event table*. Finally, the fault specification file is read, and each Boolean expression specifying a global state in which a fault should be injected is converted into an efficient internal form.

After initialization, each runtime can A) process notifications that come from its probe, B) change the state of its state machine based on these notifications, C) notify other state machines of local state changes of which they should be notified, and D) request that its probe inject faults when its perceived global state matches a fault expression.

After the initialization is complete, two extra threads are created, in addition to those used by the application and probe. The first thread blocks on the sockets waiting for remote state change notifications. When a remote notification arrives, the state entry in the node table is changed for the node that sent the notification. These notifications thus update the runtime's view of global state at a point determined by the state machine specification of the sending node. The second thread contains the fault parser that observes the perceived global state of the system to determine (based on the Boolean expressions read in from a fault specification file) whether a fault should be injected in the current state. If a Boolean expression is true, it injects the fault and records the time of injection in the local event timeline. The fault parser also terminates the experiment when all state machine nodes reach a special, predefined "EXIT" state. When this occurs, the fault parser ends the experiment, and, if appropriate, the next experiment in the study begins. Local state changes are induced by notifications from the probe that cause the runtime's state machine to look up the event, record the new local state in the appropriate position in the node table, send remote state notifications if needed, and record the state transition in the local event timeline. In this way, the partial view of the global state for a node is maintained and used to make decisions regarding fault injections. When all the experiments associated with one study have been executed, the user may select the next study to be run.

3.3 Calculation of a Single Global Event Timeline

The local event timelines obtained using the approach described in the last subsection can be used, together with the bounds obtained for $\alpha_{i,j}$ and $\beta_{i,j}$, to obtain physical bounds on the occurrence time of each event (state change or fault injection) that occurred during an experiment. As mentioned earlier, this important step allows us to determine whether the injection of a fault during an experiment was performed as intended. How these bounds on event occurrence times are determined is described below.

In particular, assume that node r is the reference node. Thus, for node r, $\alpha_r^+ = \alpha_r^- = 0$ and $\beta_r^+ = \beta_r^- = 1$. When exchanging messages with processor j, equation (2) becomes:

$$C_j(t) = \alpha_j + \beta_j C_r(t) + \delta_{r,j} \tag{3}$$

Projected on the timeline of the reference processor, this becomes $C_r(t) = \dfrac{C_j(t) - \alpha_j - \delta_{r,j}}{\beta_j}$. An event observed at time $t = T$ on processor j thus corresponds to an event occurring between the bounds $C_r^-(T)$ and $C_r^+(T)$ on the reference processor timeline defined by:

$$C_r^-(T) = \frac{C_j(T) - \alpha_j^+ - \delta_{r,j}}{\beta_j^-} \approx \frac{C_j(T) - \alpha_j^+}{\beta_j^-}$$ and

$$C_r^+(T) = \frac{C_j(T) - \alpha_j^- - \delta_{r,j}}{\beta_j^+} \approx \frac{C_j(T) - \alpha_j^-}{\beta_j^+}.$$

Since $\delta_{r,j} = \delta_j - \delta_r$, $\delta_j = \gamma_j + e_j$, and $|\gamma_i| << |e_i|$, we can assume that, as in [Jezequel 96], $|\delta_{r,j}| = |\delta_j - \delta_r| \approx 2\max(|e_j|) \approx g$, with the resolution of the clock assumed to be the same for all processors. The computers used in this paper for fault injection have a resolution of $g \approx 10^{-8}$ s, and $\delta_{r,j}$ can thus be neglected in the expressions of $C_r^-(T)$ and $C_r^+(T)$.

3.4 Determination of Whether Faults Were Injected Correctly

Since only experiments in which faults are injected properly are desired, the experiments are checked after execution to determine whether the specified fault(s) were injected in states as intended. If not, the experiment is dis-

carded. Placing all events on a single global timeline, using the bound calculated in the last subsection, allows us to determine whether faults were injected properly. First, using the constructed global timeline, the bounds on injection time for each fault are determined. The checker then determines whether this interval falls entirely within the interval in which the system was in the (global) state in which the fault was to be injected. This interval is defined by the greatest lower bound and the least upper bound of the individual node state entrance and exit times, respectively. More specifically, the upper bound of the state start time and lower bound of the fault injection time are used to determine whether the fault was injected after the state was entered. Likewise, the lower bound of the state end time and upper bound of the fault injection time are used to determine whether the fault was injected before the state was exited. If both the criteria are met, the fault was injected as intended. Note that even if both criteria are not met, it may be the case that the fault was injected correctly, but Loki assumes that it was not, to be sure that no experiments with incorrect fault injections are mistakenly deemed to be correct. This procedure is repeated for each injection that should have been made in the experiment; the experiment is only marked as successful if all the injections in the experiment were done correctly.

Figure 2 shows some possible fault injection times with respect to the state in which the fault was to be inserted. In this figure, each arrow marked "enter state upper bound" is the maximum of the upper bounds of the individual local state entrances. Likewise, each arrow marked "leave state lower bound" is the least of the individual state lower bounds on state exit time. Among the cases shown in Figure 2, Loki considers only case 3 a valid fault injection, according to the rules described in the last paragraph.

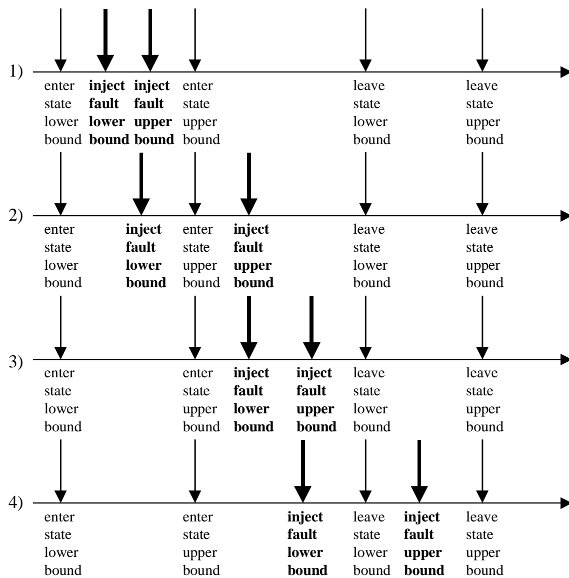

Figure 2. Possible Fault Injection Scenarios

4. Experimental Results

A prototype implementation of Loki has been constructed to test our approach. The implementation runs on Linux (Redhat 5.2) machines, and our testbed consisted of three 233 Mhz Pentium II machines connected by two 100 Mb Ethernets. One Ethernet was used for the application, and the other was used to transmit Loki state machine notifications. As implemented, Loki takes a set of state machines and fault specifications and conducts a fault injection campaign according to these specifications. The output of the tool is a single global timeline for each experiment, along with an indication of whether the intended fault(s) for the experiment were injected correctly. The test application implements a very simple leader election protocol. The application consists of three processes, which run on three different nodes. These nodes (and their associated state machines) are named "sylvester," "heathcliff," and "persian." In this protocol, the three processes elect a leader from amongst themselves. To do this, each process chooses a number and sends it to the remaining two 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.

4.1 State Machine Specification

For the test application, all the state machines have the same architecture. This common architecture is shown in Figure 3. The nodes in the graph are labeled with state names corresponding to phases of the election application, and the arcs are labeled with probe notifications, which are sent from each probe to indicate that the node in question has entered a new phase of the protocol. Figure 4 gives, for each state machine, and each state, the list of other machines that should be notified when that state is entered. Note that these notifications are not identical for all state machines, since the notifications needed depend on the Boolean expressions specifying the faults to be injected at each node.

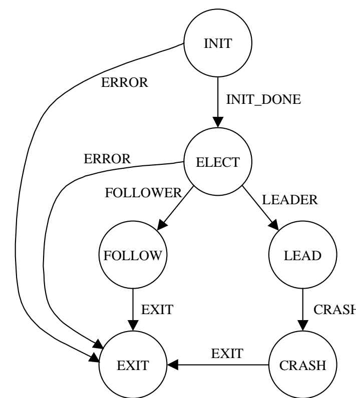

Figure 3. State Machine for Each Node

At the start of the application, each state machine is initialized to the INIT state by the associated probe. The INIT state represents the initialization of the processes, which involves setting up communication sockets. If an error occurs during initialization, the probe sends an ERROR notification to the state machine, which then makes the transition to the EXIT state. Otherwise, after the socket setup, 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. An error during election flags an ERROR and the state machine enters the EXIT state. If an error does not occur, the state machine associated with the leader goes through the CRASH state to the EXIT state, while the remaining state machines go to the EXIT state. The EXIT state marks the end of a process's execution.

STATE	*sylvester*	*persian*	*heathcliff*
INIT	heathcliff	heathcliff	sylvester
ELECT	persian, heathcliff	sylvester, heathcliff	persian, sylvester
FOLLOW	-	-	-
LEAD	-	-	-
CRASH	-	-	-
EXIT	persian, heathcliff	sylvester, heathcliff	persian, sylvester

Figure 4. Notify Lists for Each of the State Machines

4.2 Fault Specification

Five different faults were injected in the election application: two in sylvester, two in heathcliff, and one in persian. The specification of these faults is shown in Figure 5.

Node name	Fault name	Boolean expression	Freq.
sylvester	sfault1	((sylvester:INIT)& (heathcliff:INIT))	Once
sylvester	sfault2	((sylvester:ELECT)& (heathcliff:ELECT)& (persian:ELECT))	Once
heathcliff	hfault1	((sylvester:INIT)& (heathcliff:INIT)& (persian:INIT))	Once
heathcliff	hfault2	((sylvester:ELECT)& (heathcliff:ELECT))	Once
persian	pfault1	((sylvester:ELECT)& (heathcliff:ELECT))	Once

Figure 5. Fault Specifications

As mentioned earlier, the Boolean expression specifies when the fault is to be injected. For example, the fault "sfault1" should be injected only when the state machines sylvester and heathcliff are both in the state INIT. The qualifier "once" specifies that the fault should be injected only the first time the Boolean expression is true, while the qualifier "always" specifies that the fault should be injected whenever the Boolean expression is true.

4.3 Results from Fault Injection

Fault injection result analysis: Figure 6 shows a portion of the output file of an experiment. The experiment consisted of injecting sfault2, defined to be injected when the three state machines are in the ELECT state. The first column of the file indicates the name of the state machine. The second column is a flag to differentiate a fault injection (flag=1) from a state (flag=0). The flag in the third column indicates the lower bound ("l") and the upper bound ("h") of the event consisting of the entrance of a state or the injection of a fault defined in column 4. The fifth column is the time in clock ticks associated with the event. We used the hardware clock on the Pentium II processor, accessible by the `rdtsc` instruction, for all timings. Since the experiments were run on a set of 233 MHz machines, each clock tick corresponds to 4.29 nanoseconds. The last column gives the offset time, in milliseconds, since the occurrence of the first event in the experiment. We can see that the upper and lower bounds for events occurring on the reference node (here, heathcliff) are identical, since our algorithm uses this clock as a reference, and computes bounds for other nodes in terms of this node's clock. Moreover, the bounds for the other events are very close and much smaller than one millisecond. This bodes well for the ability of our algorithm to determine whether faults were injected correctly. Finally, note that the injection of sfault2 occurred when the three state machines were in state ELECT; thus, sfault2 was properly injected.

```
persian 0 l INIT_LEAF 48186103765 0.000000
persian 0 h INIT_LEAF 48186103842 0.000077
sylvester 0 l INIT_LEAF 48186246754 0.142989
sylvester 0 h INIT_LEAF 48186246782 0.143017
heathcliff 0 l INIT_LEAF 4818679543 0.691666
heathcliff 0 h INIT_LEAF 48186795431 0.691666
heathcliff 0 l ELECT 48421412646 235.308881
heathcliff 0 h ELECT 48421412646 235.308881
persian 0 l ELECT 48656277005 470.173240
persian 0 h ELECT 48656277500 470.173735
sylvester 0 l ELECT 48656278777 470.175012
sylvester 0 h ELECT 48656279316 470.175551
sylvester 1 l sfault2 48658583337 472.479572
sylvester 1 h sfault2 48658583878 472.480113
heathcliff 0 l FOLLOW 48659144990 473.041225
heathcliff 0 h FOLLOW 48659144990 473.041225
heathcliff 0 l EXIT 48659155725 473.051960
heathcliff 0 h EXIT 48659155725 473.051960
```

Figure 6. Portion of an Output File

Results for the multiple injected faults: Figure 7 shows the results obtained for several injections of the five faults

specified in Figure 5. The injection can result in a proper fault injection ("P" in Figure 7) or an improper injection ("I"). In some cases, the fault could not be injected ("N"). This could happen if, for example, a machine never reached the injection state, as perceived by the state machine in the node. Finally, some experiments failed because of problems related to the operating system. We observe that even if most injections lead to the same result for different experiments, some results vary from one experiment to another.

Experiment Number	hfault1	hfault2	sfault1	sfault2	pfault1
1	P	N	P	N	I
2	N	N	N	N	I
3	P	N	P	N	I
4	P	N	P	N	I
5	N	N	I	N	I

Figure 7. Results for the Five Injected Faults

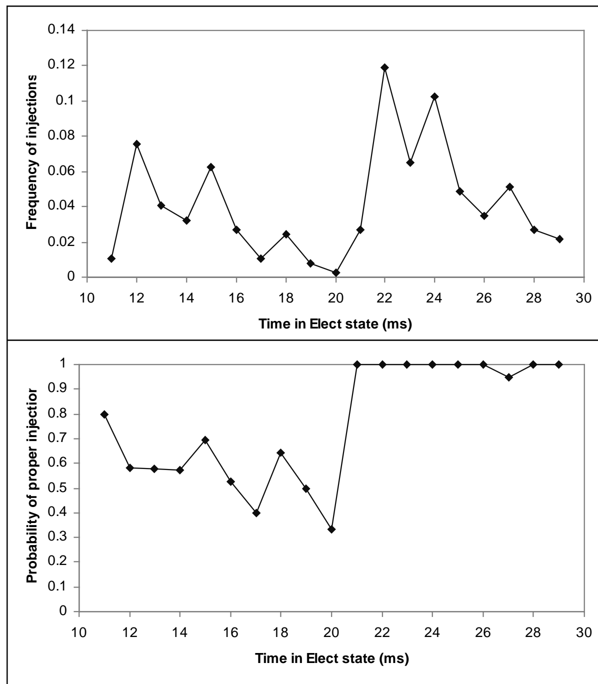

Figure 8. Correct Fault Injection Probability as a Function of the Time Spent in the ELECT State (Standard Linux Kernel – 10 ms Timeslice)

Proper fault injection trends for sfault2: Next, we focus on the injection of fault sfault2. sfault2 is a fault injected in node sylvester when the three state machines are all in the ELECT state. The goal in this case is to evaluate the probability of proper injection as a function of the time spent in the state where the injection occurs. We have thus introduced artificial delays in the ELECT state (using the `usleep()` system call) and have analyzed results for the standard version of Linux (Figure 8) and for a modified version in which the timeslice of the scheduler is 1 millisecond instead of 10 milliseconds (Figure 9).

More specifically, to change the time that the three machines spend in the ELECT state, we introduced delays when each state machine reaches the ELECT state. In particular, we added delays ranging from 1 millisecond to 20 milliseconds, and ran a total of 400 experiments for each operating system timeslice setting, with twenty experiments being run for each delay considered. We then observed, for each experiment, the length of time that all three state machines are in the ELECT state. These times were clustered in classes of 1 millisecond. The top parts of Figures 8 and 9 show the fraction of experiments in each time cluster. Note that in the case of an operating system timeslice of 10 milliseconds, the actual time spent with all state machines in the ELECT state is not uniformly distributed, even though the additional delays introduced were. We believe that this is due to the effects caused by the OS scheduler, which may not always schedule the Loki runtime immediately, even though it is ready to be run.

Comparing Figures 8 and 9, we can make the following observations. Even if certain of the delays introduced in the ELECT state of each state machine were smaller than 10 milliseconds, Figure 8 shows that the time that the three state machines spent in the ELECT state was always higher than 10 milliseconds. This is due to the timeslice of the scheduler of the standard version of Linux. By decreasing the value of the timeslice, we obtain a very different distribution of time spent in the ELECT state. In that case, as shown in Figure 9, the time that the three state machines spend in the ELECT state matches much more closely the delays introduced in each state machine.

The probability of proper injection is also very different for the two timeslice settings. In particular, Figure 8 shows that for the standard Linux scheduler, the probability of correct injection varies, but is around 0.6 for ELECT state sojourn times less than 20 milliseconds. For higher ELECT state sojourn times, the probability of correct execution is nearly 1. The timeslice of 10 milliseconds is again responsible for the shape of the curve. This is particularly obvious because the change in the value of the probability occurs exactly at a multiple of the timeslice value. Figure 9 shows, on the other hand, a rapid increase of the probability of proper injection and a fluctuation between 0.9 and 1. The impact of the timeslice value is thus no longer the dominant factor in determining whether an injection is correct for the range of time studied.

5. Conclusions

This paper presents a new approach to fault injection in distributed systems that makes possible the injection of faults based on the partial view of the global state of a distributed system. In doing so, it facilitates fault removal, by

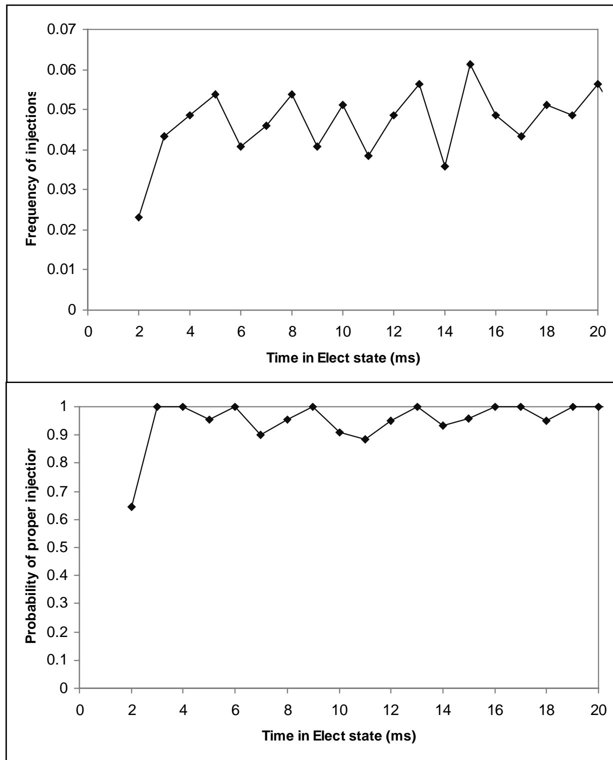

Figure 9. Correct Fault Injection Probability as Function of the Time Spent in the ELECT State (Linux Kernel – 1 msec Timeslice)

identifying the particular global state/fault combinations that lead to failures, and dependability assessment, by permitting faults to be correctly injected in representative system states. It does this by 1) providing a mechanism for recording and sharing a partial view of the global state between nodes in a distributed system, and 2) developing and applying a theory to determine whether a fault injection that depends on the partial view of the system global state was done correctly. Together, these two capabilities give us the ability to inject correlated faults into a distributed system efficiently and predictably. To the best of our knowledge, it is the first successful attempt to provide this functionality. As shown by the experimental results that we have obtained, we can successfully inject an increasing percentage of intended faults as the time spent in a chosen state increases. More importantly, we can identify the experiments that led to successful injections, and use these experiments in fault removal and/or dependability assessment activities.

We are currently implementing a graphical user interface for specifying state machines, faults to inject, measures to compute, and other information regarding a particular campaign. We are also developing a new language to specify a large range of measures, and developing statistical methods to provide point and interval estimates for use in dependability assessment. When complete, this effort will result in complete, easy-to-use tool for experimentally validating dependable distributed systems.

Acknowledgments

We would like to thank Ryan Lefever for his assistance in the construction of the graphical user interface of Loki.

References

[Alvarez 95] G. Alvarez and F. Cristian, "A centralized failure injection environment for the validation of distributed fault-tolerant protocols," Univ. of Calif. at San Diego, La Jolla, CA, Tech. Rep. CS95-458, 1995.

[Bhatt 95] 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 Int. Parallel Proc. Symp.*, 1995, pp. 494-501.

[Cristian 89] F. Cristian, "Probabilistic clock synchronization," *Distributed Computing*, vol. 3, pp. 146-158, 1989.

[Dawson 96] S. Dawson, F. Jahanian, and T. Mitton, "ORCHESTRA: A fault injection environment for distributed systems," Univ. of Michigan, Ann Arbor, MI, Tech. Rep. CSE-TR-298-96, 1996.

[Duda 87] A. Duda, G. Harrus, Y. Haddad, and G. Bernard, "Estimating global time in distributed systems," in *Proc. of the 7th Int. Conf. on Distributed Computing Systems*, 1987, pp. 299-306.

[Echtle 92] 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 Distributed Systems*, 1992, pp. 28-35.

[Ellingston 73] C. E. Ellingston and R. J. Kulpinski, "Dissemination of system time," *IEEE Trans. on Comm.*, vol. COM-21, pp. 605-623, May 1973.

[Han 95] 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 Int. Computer Performance and Dependability Symp.*, 1995, pp. 204-213.

[Henke 98] D. Henke, "Loki – An empirical evaluation tool for distributed systems: The experiment analysis framework," M.S. thesis, Univ. of Illinois, Urbana, IL, 1998.

[Hofmann 88] R. Hofmann, R. Klar, N. Luttenberger, B. Mohr, and G. Werner, "An approach to monitoring and modeling of multiprocessor and multicomputer systems," in *Proc. of the Int. Seminar on Performance of Distributed and Parallel Systems*, 1988, pp. 91-110.

[Jezequel 96] J.-M. Jezequel and C. Jard, "Building a global clock for observing computations in distributed memory parallel computers," *Concurrency Practice & Experience*, vol. 8, no. 1, pp. 71-89, Jan.-Feb. 1996.

[Kopetz 87] H. Kopetz and W. Ochsenreiter, "Clock synchronization in distributed real-time systems," *IEEE Trans. on Comp.*, vol. C-36, pp. 933-940, Aug. 1987.

[Lamport 78] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, pp. 558-565, July 1978.

[Lamport 85] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *Jour. of the ACM*, vol. 32, pp. 52-78, Jan. 1985.

[Maillet 95] E. Maillet and C. Tron, "On efficiently implementing global time for performance evaluation on multiprocessor systems," *Jour. of Parallel and Distributed Computing*, vol. 28, pp. 84-93, July 1995.

[Mattern 89] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. of the Int. Workshop on Parallel and Distributed Algorithms*, 1989, pp. 215-226.

[Mink 90] A. Mink, R. Carpenter, G. Nacht, and J. Roberts, "Multiprocessor performance-measurement instrumentation," *Computer*, vol. 23, pp. 63-75, Sept. 1990.

[Pistole 98] J. Pistole, "Loki – An empirical evaluation tool for distributed systems: The run-time experiment framework," M.S. thesis, Univ. of Illinois, Urbana, IL, 1998.

[Stott 99] D. T. Stott, Z. Kalbarczyk, and R. K. Iyer, "Using NFTAPE for rapid development of automated fault injection experiments," in *Digest of FastAbstracts of the 29th Ann. Int. Symp. on Fault-Tolerant Computing (FTCS-29)*, Madison, Wisconsin, USA, pp. 39-40, June 1999.

[Sedgewick 84] R. Sedgewick, *Algorithms*. Reading, MA: Addison-Wesley, 1984.