Loki: A State-Driven Fault Injector for Distributed Systems [*]

Ramesh Chandra, Ryan M. Lefever, Michel Cukier, 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
{*ramesh, lefever, cukier, whs*}*@crhc.uiuc.edu*

Abstract

Distributed applications can fail in subtle ways that depend on the state of multiple parts of a system. This complicates the validation of such systems via fault injection, since it suggests that faults should be injected based on the global state of the system. In Loki, fault injection is performed based on a partial view of the global state of a distributed system, i.e., faults injected in one node of the system can depend on the state of other nodes. Once faults are injected, a post-runtime analysis, using off-line clock synchronization, is used to place events and injections on a single global timeline and to determine whether the intended faults were properly injected. Finally, experiments containing successful fault injections are used to estimate the specified measures. In addition to reviewing briefly the concepts behind Loki and its organization, we detail Loki's user interface. In particular, we describe the graphical user interfaces for specifying state machines and faults, for executing a campaign, and for verifying whether the faults were properly injected.

Keywords : *Distributed system validation, Experimental evaluation, Fault injection, State-driven fault injection.*

1. Introduction

The increasing use of distributed systems to build critical applications motivates the development of techniques to validate their dependability. Fault injection is an important and effective way to validate such systems. However, fault injection of distributed systems is a difficult and challenging task. The reasons for this are as follows. The behavior of a distributed system depends on the state changes and events occurring in the system's different processes. Thus, the faults occurring in such a system can depend on its global state. This necessitates that, while injecting faults in a system, the fault injector keep track of its global state to inject realistic faults and/or errors. One approach would be to synchronize at all state changes in the system, but this would be far too intrusive, and might affect the behavior of the system in unacceptable ways. The alternative to synchronization is to use state change notifications to keep track of the global state. Though these notifications are less intrusive than synchronization, fault injections based on notifications could occur in improper states, since the system could change state between notifications. Measurements based on such improper injections are not valid.

A fault injector for distributed systems should thus be able to inject faults based on the global state of the system, and at the same time not be too intrusive to the system under study. It should also be able to determine whether a particular set of faults were injected as intended, so measures can be calculated using only the intended injections. With these issues in mind, we have developed a global state-driven fault injector, called *Loki*, for distributed systems. Loki injects faults in a distributed system based on a partial view of its global state obtained using notifications, and can determine, using a post-analysis, whether each fault was injected as intended. In this paper, we describe the features of the Loki fault injector and how to use it for conducting a fault injection campaign. We also provide an overview of Loki concepts and the Loki runtime architecture.

Other fault injection and measurement tools, including EFA [6], Orchestra [5], SPI [2], NFTAPE [8], DOCTOR [7], and CESIUM [1], have also focused on distributed systems. These tools work well for their intended purposes. However, Loki is unique in that it supports fault injections based on the global state of the system combined with a powerful language for defining measures. For a comparison of these tools with Loki, see [4].

The remainder of the paper is organized as follows. Section 2 presents a brief description of the concepts underlying Loki, namely, partial view of the global state, and off-line

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

clock synchronization. Section 3 provides an overview of the Loki runtime architecture. Sections 4, 5, and 6 illustrate the use of Loki for performing a fault injection campaign. In particular, Section 4 describes the specification of a campaign using the Loki interface; Section 5 details the execution of a campaign using the Loki interface; and Section 6 explains off-line analysis of the execution results and how to obtain the required measures from them. Finally, Section 7 presents our conclusions.

2. Review of Loki concepts

In this section, we briefly review the basic concepts of Loki, namely, the partial view of the global state and off-line clock synchronization. Additionally, we introduce various terms that are used in the rest of the paper. More details of what is presented here can be found in [4].

The concept of *state* is fundamental to Loki. We assume that at the desired level of abstraction (for fault injection), the execution of a component of the distributed system under study can be specified as a state machine. The global state of the system is the vector of the local states of all of its components. During the fault injection process, it may be necessary to inject faults in a component based on the state of other components of the system. It can be seen that to do this, it is not necessary to keep track of the complete global state of the system at all times; instead, it is sufficient to track an "interesting" portion of the global state that is necessary for the injection of the required faults. This interesting portion is called the *partial view of the global state*, and its selection depends on the particular system under study and the faults to be injected.

In Loki, the distributed system (under study) is divided into basic units (i.e., processes) from each of which state information is collected and into each of which faults are injected. Such a basic unit of the distributed system along with the Loki runtime attached to it is called a *node*. The Loki runtime maintains the partial view of the global state for each of these nodes and injects faults in them when necessary. It also records state changes and fault injections along with their occurrence times. The runtime only uses the necessary *state change notifications* between nodes to keep track of the partial view of the global state. Also, to be as non-intrusive to the system as possible, the runtime does not block the system while these notifications are in transit. This means that the system could change state while the notification is in transit, implying that the partial view could sometimes be out-of-date. This could lead to incorrect fault injections and hence incorrect measures.

To avoid such errors, Loki performs a post-runtime check on every fault injection to determine whether it has indeed been performed in the desired state (an off-line check is used to avoid the expense and intrusiveness of an on-line check). Only the correct fault injections are then used in computing the measures. The post-runtime check involves placing the local times from each of the nodes into a single global timeline and then determining whether the fault was injected in the right state. Loki uses an *off-line clock synchronization* algorithm to translate the local times to a global timeline. Synchronization messages, which are used by this algorithm, are generated by the runtime before and after the application execution. These messages are non-intrusive, since they are generated when the application is not executing. This algorithm assumes that the drifts of the system clocks are linear. A more detailed explanation of the algorithm, along with its use in Loki, can be found in [4].

3. Overview of Loki runtime architecture

In this section, we provide a short overview of the Loki runtime architecture. The runtime executes along with the distributed system and maintains the partial view of the global state necessary for fault injections. It also performs fault injections when the system transitions to the desired states and collects information regarding state changes, fault injections, and their occurrence times.

As shown in Figure 1, there is a Loki runtime for each of the nodes in the distributed system. The runtime can be divided into two main parts: one that is independent of the system under study and one that is dependent on it. The *state machine*, *state machine transport*, *fault parser*, and *recorder* constitute the system-independent part, while the *probe* is the system-dependent part. The state machine keeps track of the partial view of the global state necessary for its node. It receives local state change notifications from the probe, and state change notifications of remote nodes from remote state machines. The state machines of different nodes send state change notifications to each other using the state machine transport. The recorder records the state changes and fault injections along with their times of occurrence. Boolean fault expressions are used to trigger fault injections. The fault parser parses these fault expressions on every state change and instructs the probe to inject the corresponding fault when an expression is satisfied. The probe in Loki has to be implemented by the system designer. The designer can either select a probe among the pre-implemented probes in Loki, or develop his/her own probe. Therefore, the designer will have considerable freedom in selecting the type of faults to inject into the system. The probe monitors the local node for state transitions and notifies the state machine of them. Also, it is the probe that performs the actual fault injections when the fault parser instructs it to do so. For more details regarding the runtime, its components, and their functions, refer to [4].

The evaluation of a system using Loki can be divided

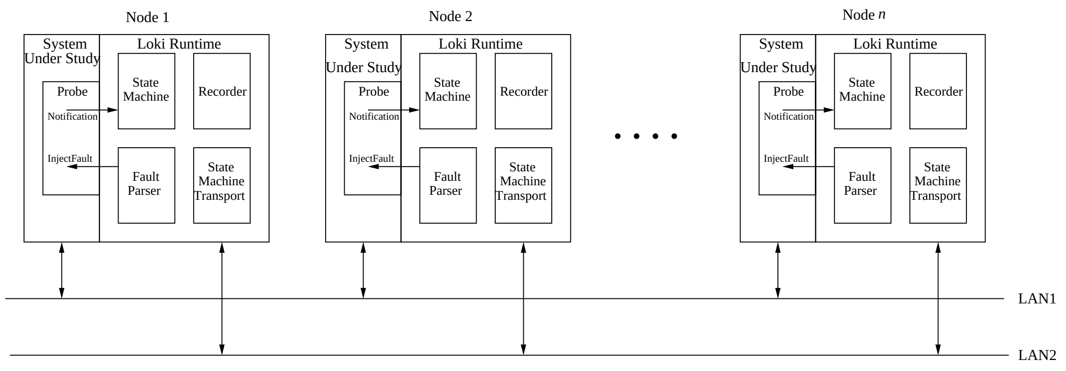

Figure 1. Loki runtime architecture

into five main phases, namely,

1. An initial synchronization-message-passing phase,
2. A fault injection and observation collection phase,
3. A second synchronization-message-passing phase,
4. Determination of experiments with properly injected faults, and
5. Computation of measures using these experiments.

In the remainder of this paper, we show how a system designer can use Loki to evaluate a system, by providing information about the five phases of evaluation in Loki. More specifically, we describe specification and execution of fault injection campaigns and off-line processing using the Loki graphical user interface.

4. Campaign specification

Loki is based on the concept of a *fault injection campaign*. A fault injection campaign for a distributed system is made up of one or more *studies*. At the study level, the system is described using state machines, which are defined by a state machine specification and a fault specification. Each study consists of a set of *experiments*, each of which is one run of the distributed application along with the fault injections corresponding to the study. Campaigns can be defined using Loki's graphical user interface, the *Loki interface*.

The Loki Manager, shown in Figure 2, controls the main functionality of the Loki interface. The panel of buttons along the bottom of the Loki Manager allows a campaign to be imported, exported, created, deleted, or copied. After a campaign is created, it can be specified with the "Edit Campaign" button. This launches a Campaign Manager window, which is similar to the Loki Manager, using which studies for the selected campaign can be created, deleted, copied, or edited. If a study is edited, a Study Manager is launched, which is also similar to the Loki Manager. Three parameters

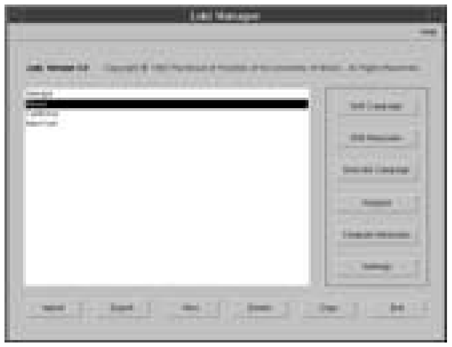

Figure 2. The Loki manager

of the study must be specified in the Study Manager: the number of experiments in the study, the time between experiments, and the application timeout. The Study Manager also allows state machines to be created, deleted, copied, and edited. After the state machines for a study have been created, each state machine must be defined with a state machine specification and a fault specification, and the distributed system must be instrumented for the study.

4.1. State machine specification

State machine specification is done using the State Machine Editor which can be seen in Figure 3, and is launched from the Study Manager. The State Machine Editor's "File" menu allows the user to save and close the editor. The "Edit" menu provides undo functionality for the state machine canvas and the ability to set the state machine's properties. The third menu item is the "Panel" menu; and it determines the control mode to be used when editing on the state machine canvas. The last menu item is the "Panel Size" menu, which allows the size of the canvas to be increased or decreased.

The three basic components that can be placed on the state machine canvas are states, event-triggered state transi-

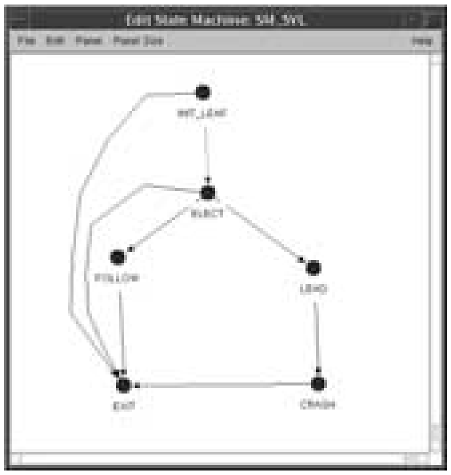

Figure 3. The state machine editor

tions, and comment boxes. States are represented on the canvas by named nodes. The state transitions are represented by connection lines between two states. The connection lines are associated with events that trigger the transitions. The comment boxes are text areas that can be used to note additional information about the state machine. The type of components that are placed on the canvas is determined by the control mode that is selected from the "Panel" menu, e.g., in the "State" mode, states can be placed on the canvas. When a state is created, it is given a name, and a list is designated of state machines that should be notified when the state machine enters the state. This list helps the state machines to maintain the partial view of the global state. When a state transition is created, the event that triggers the transition must also be provided by the user.

4.2. Fault specification

After the state machines have been specified, the fault specifications should be defined. The Fault Specification Editor is launched from the Study Manager at the same time as the State Machine Editor. It shows a listing of all the faults that should be injected into the state machine's corresponding application. The fault specification of each fault consists of three parameters. They are the fault name, a fault expression that determines when the fault should be injected, and an indication of whether the fault should be injected only the first time that the expression is true, or every time that the expression becomes true. As described earlier, the actual fault injection code is called a "probe." The method for specifying a probe is given in Section 4.3.

The main part of the fault specification is the fault expression that triggers the fault. The expression represents some partial view of the global state in the distributed system. The variables in the fault expression are (state machine:state) pairs. The expressions make use of '&' for an AND, '|' for an OR, and '˜' for a NOT. An example expression is:

```
   ((StateMachine1:State5)&(˜(StateMachine3:
State3)))
```

This indicates that a fault should be injected when StateMachine1 is in State5 and StateMachine3 is not in State3.

4.3. Instrumenting the distributed system

After the state machine and fault specifications in a study have been defined, the application corresponding to each state machine must be instrumented. The Loki runtime is implemented in C++; therefore, the instrumented portion must also be programmed in C++. There are three steps to instrument an application:

- Probe Implementation
- Event Notification
- Use of `appMain()`

A probe must be defined for the application, and its `injectFault()` function must be implemented. This function takes in a fault name (which corresponds to the name given in the fault specification), implements injection of the fault, and returns the time when the fault is actually injected. By having the Loki user write the `injectFault()` code, Loki is capable of providing a high degree of freedom in the types of faults injected. The probe should also indicate to the state machine the occurrence of events that were specified in the state machine specification. This is done using the `notifyEvent()` method of the state machine and passing the event name and the time at which the event occurred as parameters. Another requirement is that `appMain()`, instead of `main()` must be used to start the application.

The instrumentation can be performed both when the application's source code is available and when it is not. If it is available, then the probe can be integrated into the application code. `notifyEvent()` and `injectFault()` can directly be a part of the application's source code. For this to be done, the application's `main()` function should be renamed to `appMain()`. If the source code is not available, the probe can be used as a monitor for the application. The probe's `appMain()` function is used to start the application, and it can monitor the application's input and output for events. These events are then communicated to the state machine using the `notifyEvent()` method. The `injectFault()` method can inject faults from outside the application. The first approach provides more accurate event notification and fault injection, but it is more intrusive than the second approach.

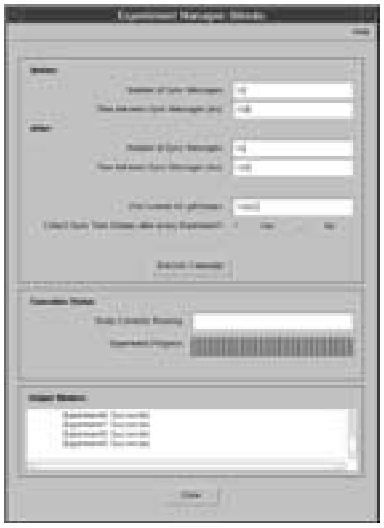
Figure 4. The experiment manager

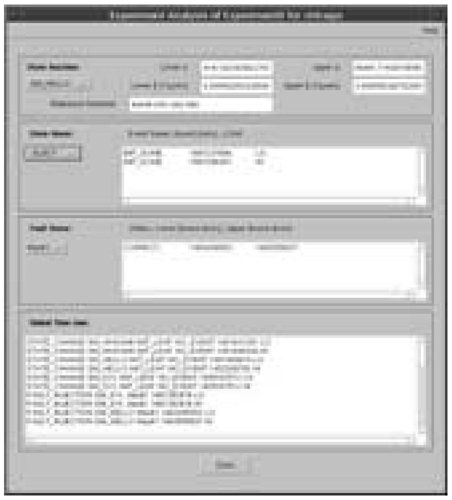
Figure 5. Analysis window for an experiment

5. Campaign execution

After a campaign is fully specified, it can be executed. To do this, the Experiment Manager, shown in Figure 4, is launched from the Loki Manager. There are six parameters in the top panel of the Experiment Manager that must be set before a campaign is executed. The Loki runtime passes synchronization messages for the off-line clock synchronization as discussed in Section 2. These messages are passed between all the hosts in the distributed system. The first two parameters represent the number of synchronization messages that should be passed before each experiment, and the delay between those messages. The next two parameters are similar, but correspond to synchronization messages that are passed after experiments. The fifth parameter indicates the port number that the synchronization messages should be passed on. The last parameter specifies whether the synchronization messages are to be passed after each experiment, or after each study.

Once all the parameters are set, the campaign can be executed with the “Execute Campaign” button. It executes each study one after another. Within each study it executes the specified number of experiments, passing synchronization messages as indicated above. During execution, the middle panel of the Experiment Manager indicates the current study and displays the experiment’s progress. The bottom panel indicates whether studies and experiments were successfully executed.

6. Off-line processing

Off-line processing consists of two steps: campaign analysis and measure estimation. The campaign analysis creates global timelines for the different experiments, and determines whether the faults were properly injected. Measure estimation in Loki uses the results of these proper fault injections to calculate statistically representative measures.

6.1. Campaign analysis

Two windows are used during campaign analysis: “Analysis,” which performs analysis computations for a campaign and displays preliminary results for a study, and “Experiment Analysis,” which shows the details for a particular experiment. The conversion to global timeline is done only once for each study. When the analysis is run for the first time, the “Analysis” window is disabled, and a dialog box shows the progress of the conversion to a global timeline. If the conversion to the global timeline has already been performed, the “Analysis” window allows the user to select a study in order to obtain a report of how many faults were correctly injected, incorrectly injected, and not injected at all for each state machine in the study. The user can also focus on one particular experiment of the study by clicking the “Examine Analysis” button.

The “Experiment Analysis” window, shown in Figure 5, presents detailed information on each experiment. The top part of the window gives the clock synchronization results for the different machines. For the meaning of these results, refer to [4]. The second part of the window shows information on each state of the selected state machine. Each entry consists of the event name, the time in clock ticks, and an indication of whether the time is an upper or lower bound of the event. Note that a time instant on a local timeline, when projected to the global timeline, becomes a time interval defined by a lower and upper bound. Hence, there are two entries for each event. The third part of the window shows all the occurrences of a fault in the selected state machine. For each fault occurrence, there is an indication of whether the fault was injected properly, not injected properly, or not injected at all. The two bounds related to each fault are also shown. The last part of the window shows the global timeline for the given experiment. The first column indicates the event type (state transition or fault injection). The second column shows the state machine in which the event occurred. If the event is a state transition, the third column indicates the ending state of the state change, and the fourth column indicates the event that triggered the transition. If a fault was injected, the third column contains the fault name. For both cases, the next column shows the time in clock ticks, and the last column indicates whether the time shown is a lower or upper bound.

6.2. Measure estimation

The goal of measure estimation in Loki is to provide a mechanism to obtain statistically representative measures which are interesting to the user. Loki uses a flexible language to describe these measures. The definition of measures is done at two levels, namely, at the study level and at the campaign level.

Each of the measures specified at the study level consists of an ordered sequence of (subset selection, predicate, observation function) triples, and is defined for a particular study. The *subset selection* is used to select a subset of experiments based on the observation function outcomes of the previous triple. Note that the subset selection of the first triple would select all the experiments of the study. The *predicate* is a Boolean expression containing queries of the form (state machine, state, time), and (state machine, start state, event, time), combined with AND, OR, and NOT. The outcome of the predicate applied to the global timeline of an experiment is called the *predicate value timeline* and is a combination of impulses and steps. The *observation function* is defined on the predicate value timeline and its outcome is called the *observation value*. The above triples are applied, in the specified order, to each of the global timelines in the study to obtain the corresponding final observation values.

Measures at the campaign level are obtained by collating the final observation values of different experiments. Depending on the method of collation, and the observations used during collation, statistically significant results such as the first four moments, and percentiles for various α-levels can be computed for these measures.

7. Conclusions

This paper describes Loki, a state-based fault injector for distributed systems. Loki is unique in its ability to inject faults based on a partial view of the global state of a distributed system and its ability to check afterwards whether the faults were correctly injected. Moreover, Loki provides a powerful language to define sophisticated measures. We have shown in this paper how a user can, by using the Loki interface, specify the fault injection campaign and the faults to be injected, execute a campaign, and analyze the obtained results to verify whether the faults were properly injected. Preliminary performance results using a simple example [4] showed that, depending on the application and the OS, high probabilities of correct fault injection can be expected for states in which the application will remain for at least several OS time-slices.

References

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

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

[3] R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders. *Loki User's Manual – version 1.0.* Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Illinois 61801, August 1999.

[4] M. Cukier, R. Chandra, D. Henke, J. Pistole, and W. H. Sanders. Fault injection based on the partial global state of a distributed system. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 168–177, October 1999.

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

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

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

[8] 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 Annual International Symposium on Fault-Tolerant Computing (FTCS-29)*, pages 39–40, June 1999.