LOKI – AN EMPIRICAL EVALUATION TOOL FOR DISTRIBUTED SYSTEMS: THE RUN-TIME EXPERIMENT FRAMEWORK

BY

JESSICA LOUISE PISTOLE

B.S., Rice University, 1996

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

To my family

ACKNOWLEDGMENTS

There are many people whom I would like to thank for their assistance in the development of the work presented in this thesis. First I would like to thank my advisor, Professor William H. Sanders, for providing me with the opportunity, resources, and motivation to participate in his research team. Professor Sanders was always willing to offer guidance both in my research and in my other educational goals, while still allowing me to work independently. I would also like to thank Dr. Michel Cukier for his role as an additional research advisor. Dr. Cukier has been an inspiration of dedication and professional integrity throughout the project, and has always offered me a valuable different perspective on my work, while ultimately allowing me to make my own research decisions. In addition, I would also like to offer my appreciation to the members of our research team at BBN Technologies for their expertise and assistance.

My research partner, David Henke, always provided helpful advice during my work on the Loki run-time experiment framework, and was instrumental in the development of the Loki tool as a whole. Alex Williamson was integral to the maintenance of the systems on which Loki was developed and also gave useful platform-specific assistance in the development of the Loki software. Jenny Applequist contributed invaluable technical editing assistance for this thesis as well as other papers and presentations associated with the project. I would also like to thank the rest of Professor Sanders's research group, especially Chetan Sabnis, for their constant support during my work.

Finally, I would like to thank those who funded the work presented in this thesis: the Defense Advanced Research Projects Agency Information Technology Office through contract F30602-96-C-0315, and the National Science Foundation through the Graduate Research Fellowship Program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1. Background

Dependability evaluation is an area of importance to computer system designers. All computer systems must be dependable, although the level of dependability required varies from system to system. In order to ensure that a given computer system has the level of dependability required, the system must be carefully validated. This validation procedure is a combination of system verification and system evaluation [1]. Verification is the process of determining, through methods such as testing or formal proofs, whether a system meets its functional specification. Evaluation is the process of determining how well a system meets its performance and dependability requirements once it meets its functional design specification. This thesis focuses on the development of a tool to assist in the empirical evaluation of computer system dependability.

Evaluating a system's dependability is not straightforward. Ideally, one would like to be able to run the system in all possible configurations and all possible environments for long enough to determine its dependability. Unfortunately, this is impossible if a system is to be designed and deployed within a reasonable amount of time and a reasonable budget. Instead, one should understand the properties of the system well enough to be able to use alternative approaches to determine the system's dependability. One such process is system modeling, in which a mathematical representation of the system is developed and solved in order to determine system properties. Such a model can be used to determine in a reasonable time span the behavior of the system over a long interval. In addition, a model can be used to determine the behavior of the system in various configurations and environments, which is especially useful when those configurations and environments are difficult to achieve during system evaluation (e.g., due to cost requirements).

Much of a system model can be designed with only the implementation of the system in mind. With a detailed knowledge of the algorithms and architecture employed in building the system, the components in a model as well as many of the relationships between these components can be defined. Some parameters, however, must be either estimated or determined through some external source.

Loki, described in this thesis and in [2], is intended mainly for the purpose of obtaining some of these parameters for fault-tolerant software distributed systems; however, it can also be used for more general computer systems. The values to be obtained for these systems are related to how often and how quickly the fault-tolerant system recovers when a fault actually occurs. These measures include coverage, latency, and dormancy. Coverage is “the probability of system recovery given that a fault exists” [3]. Dormancy and latency are times rather than probabilities. Dormancy is the time corresponding to “the activation of an injected fault as an error.” Latency is the time from the error to either failure of the system or detection and system recovery [4].

Since faults are generally rare, it would take too long to wait for such events to happen in the system’s actual environment. Instead, one can utilize a tool that generates faults at a much higher rate in order to determine the values of the desired measures from observations taken at the time of the fault. This process, called fault injection, can be used for two different purposes: fault removal and fault forecasting. Fault removal is useful in verification, and fault forecasting is useful in evaluation.

When using fault injection for evaluation, the user decides on a fault injection campaign, specifying different experiments to be run, the number of times each experiment is to be run, and the method for combining the measures determined from the different experiments into a global value for the system. For a given experiment, the user describes the faults to be injected and the means for computing the value of the desired measure from that experiment. Once the fault injection campaign has been specified, the fault injection tool is used to execute the experiments and determine the value of the system measures. These measures correspond to the parameters discussed previously.

Loki is an empirical evaluation tool that utilizes fault injection to generate the measures described above. Specifically, Loki must generate time and probability distributions with confidence intervals. In addition, Loki must allow the user to inject faults in specific global system states, so that more complicated experiments, such as those involving correlated faults, can more easily be designed. This facility plays an important role in system evaluation, but it also has the benefit of allowing Loki to be used as a framework for system verification.

1.2. Related Work

There are many fault injection and measurement tools already in existence, several of which are targeted specifically to distributed systems. In this section those tools that meet some of the objectives of Loki are described. None of these tools, however, meets all of the necessary requirements for system evaluation.

CESIUM [5] 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. The code of the system under study is not instrumented; therefore decisions and observations cannot be made based on the internal state of a process, and fault injections cannot be targeted to specific states of operation within a node. In addition, this simulation-based approach is not appropriate for all distributed applications.

DOCTOR [6] runs the system in its actual distributed form, with a set of control processes that run on a separate node in order to minimize their effect on the system. Faults can be injected probabilistically, or they can be injected based on system messages seen by the node; however, nodes do not communicate additional data to determine when to inject faults. Timing is done using a hardware solution, which requires a shared backplane bus not available to all systems.

EFA [7] is designed to be used for verification rather than evaluation. Fault injection in this system is limited to the data-link layer of a communicating process. EFA does allow for the exchange of information between local nodes, but it does not provide a formal means of specification of this communication, making experiment design more difficult and verification of the proper injection of faults in a certain state too complex. Additionally, EFA provides no means for getting all of the time stamps on a global timeline, thereby making it difficult to determine the values of measures if all events do not occur on one host machine.

Orchestra [8] is a more flexible fault injection tool, integrating into the system under study as a layer that can be inserted anywhere in a layered protocol stack. Orchestra also allows for fault injection based on the local state of a node; however, it does not provide a formal means for exchanging information between nodes. Additionally, since Orchestra is targeted towards verification, it does not provide a means for generating statistically significant measures.

SPI [9], a commercial approach to the evaluation problem, 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. This tool was developed primarily with measurement and evaluation in mind, as opposed to being designed specifically for fault injection; however, SPI could be used for fault injection. This system currently runs on SUN workstations and the Intel Paragon, and is being ported to other systems.

1.3. Research Goals

As discussed previously, Loki must produce statistically significant measures as well as allow the user to inject faults in specific global system states. To do this, Loki must meet the following goals:

- Provide a formal means of specification for the on-line maintenance of a view of the global state of the system.
- Provide a means for global control of the running of experiments.
- Provide a means for the user to specify fault injections that are dependent upon the global view of state.
- Provide a flexible means of communication between the fault injection mechanism and the global state view where the code is maintained.
- Provide a means for translating empirical observations taken on different host machines to a single timeline.
- Provide a means for the determination of the correctness of global-state-based fault injection.
- Provide a means for the estimation of the measures desired.

2. ARCHITECTURE OVERVIEW

2.1. Overview

This chapter presents the high-level architecture for the Loki evaluation tool. Loki is intended to meet certain goals for evaluating distributed systems. These goals are discussed in Section 2.2. Sections 2.3 and 2.4 discuss the architecture of the tool, which is composed of two major pieces. The first component, the run time, carries out the experiments and collects results. The Loki run-time architecture is covered in Section 2.3. The analysis tools, which are the second component, generate the desired results from the experiment data. Section 2.4 covers the analysis tools. Finally, Section 2.5 briefly describes the steps in evaluating a system using Loki.

2.2. Goals

As discussed in Chapter 1, validation of reliable distributed systems is both necessary and complex. Loki is designed to be used for different phases of this validation process. To accomplish this, Loki must meet at least two goals. First, the tool must be able to inject faults in specific system states, as defined by the combined state of multiple system components. This allows more complex faults to be injected, which will allow fault removal as well as fault forecasting in systems where very precise faults are likely to occur. Methods for meeting this goal are discussed in Section 2.3 and in more detail in Chapter 3. Second, the tool must provide statistically significant interpretation of the experiment results, yielding coverage and performance-related measures. Without this ability, results obtained cannot be justified. Additionally, the results cannot be used as input for system models that are meant to predict behavior in system configurations that are difficult to create using a testbed (e.g., systems with a very large number of hosts). Accomplishing this second goal requires several steps that are discussed briefly in Section 2.4 and in more detail in [2].

For the Loki empirical evaluation tool, these goals were addressed with the additional constraint that it must be possible to test systems using multiple hosts on a testbed. The alternative, simulation of multiple networked hosts on a single computer, requires specialized real-time operating systems and very detailed knowledge of the interaction of the system being evaluated with the operating system.

2.3. Run-time Architecture

One of the primary goals of Loki is the ability to conduct sophisticated experiments based on the state of multiple system components. In order to explain how Loki addresses this goal, the concept of state must first be examined more closely.

2.3.1. Local and global state in fault injection

In a truly distributed system, a given 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 adequately remove faults in a system and to evaluate a system designed to tolerate very specific faults, it may be necessary to introduce a fault in some component based on the state of one or more other components, even at a time when those states are unknown.

To inject such faults, an empirical evaluation tool needs to have a picture of the global state of the system; however, it is not necessary for the tool to know the exact state of every component at all times. Instead, it is necessary only to have *a measure-driven partial global view of state*. In other words, only the state of "interesting" components and only "interesting" state changes on those components are needed, where "interesting" means relating to the measures of interest.

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 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. Nodes can be, but are not limited to, hosts, processes, or threads. The Loki run-time architecture provides a framework for defining and maintaining a partial global view of state using a set of nodes and conducting fault injection campaigns using that partial global view of state.

2.3.2. Global view of the Loki architecture

There are many ways to track a partial global view of state in a distributed system. Loki achieves this view by employing communicating state machines. Figure 1 shows the global architecture of a system using Loki for a fault injection experiment. 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 *notifications* (messages containing information that can change the global state of the system) to subsets of all of the state machines in the system as necessary in keeping the partial global state needed for the experiment.

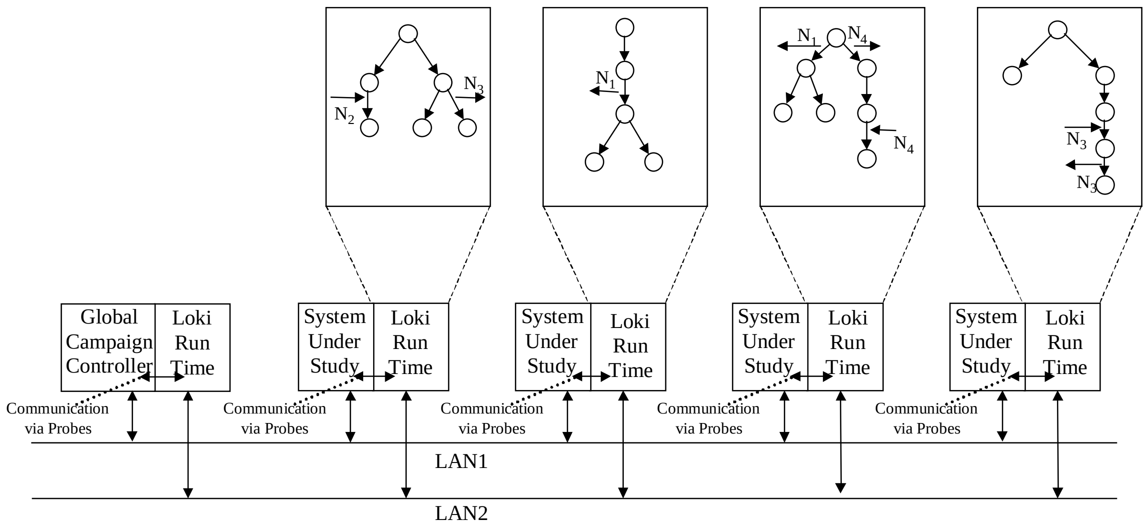


Figure 1: Global Architecture of Fault Injection Using Loki

Because an empirical evaluation tool should be designed to be nonintrusive, Loki does not block the system under study while waiting for a notification to arrive at another state machine. This means that by the time a notification reaches its target state machine(s), the system may actually have changed state again. This implies that the partial global view of 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 to transmit a notification after the state is reached. Any fault injection based on a global state of the system must be checked after the experiment is completed. More details regarding this checking are covered in Section 2.4 and in [1].

Loki also does not employ any on-line clock synchronization algorithms since on-line clock synchronization is expensive and intrusive. Additionally, on-line clock synchronization is unnecessary, since faults that are based on a "global" time can be triggered by a combination of the partial global state and a local time-keeping mechanism. Global time is necessary only for computing results, and thus is handled off-line by the analysis tools.

In addition to the communicating state machines associated with the system nodes, there is a global campaign controller responsible for initializing experiments for the fault injection campaign, determining when the experiment is complete, and terminating the campaign when the appropriate number of experiments has been run.

2.3.3. Local view of the Loki architecture

State changes in a state machine associated with a node can happen either because of a notification received from another state machine or because of some event occurring on the node iself. The instrumentation in the system node is responsible for passing these events to the run time via a standardized interface. Based on these events or on notifications received from other state machines, a state machine will transition to a new state and/or execute actions (e.g., sending a notification).

Loki provides the standardized interface to the state machine via a *probe*, which is an object containing methods for inserting events, accessing state machine information, and taking system observations. The user may choose to instrument a node in one of three ways. First, the user can utilize the basic probe provided by the Loki system as an interface only, adding any additional instrumentation in the code of the system under study. An alternative is to augment the probe that is provided with the necessary instrumentation, treating the basic probe as a template. Finally, the user can combine these two approaches, placing some of the instrumentation code inside the probe and some in the system under study. Whichever choice is made, instrumentation code should take a form suited to the system under study. Some examples of ways to use instrumentation are a layer in a protocol stack, a function call, or an object "wrapper."

The actual fault injections in an experiment are conducted in the instrumentation of the system node rather than in the core of the state machine. This allows the flexibility to inject the type of faults that are most appropriate for a given system under study. The instrumentation code "decides" when to inject faults based on feedback received from the state machine via the standardized interface and through any additional measures implemented by the user in designing an experiment.

2.3.4. Noncommunicating architecture

Because there are many fault injection experiments that do not require any notion of global state, Loki provides an additional mode of operation in which the state machines are isolated and notifications are not allowed. This allows these simpler experiments to be conducted without the additional overhead of the communicating state machines.

2.4. Analysis Tools

Loki's analysis tools have three major responsibilities. First, since there is no clock synchronization employed during the experiments, the analysis tools must place observations taken on different computers (and therefore different clocks) onto a single timeline. Second, the analysis tools must determine whether fault injections based on Loki's global state were performed correctly. Third, the analysis tools must generate the results/measures required by the fault injection campaign for experiments in which faults were properly injected.

2.4.1. Off-line clock synchronization

As discussed in Section 2.3.2, Loki does not synchronize clocks on-line. Because of this, the analysis tools are responsible for placing all recorded observations from the experiments onto a single 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 [10].

The analysis tools record message-passing times between all machines in the evaluation testbed twice, once before the experiments are run and once after all experiments are complete. Using these time stamps, the offsets and linear drifts between each machine and one machine chosen as the reference machine are calculated. Given these values, the local times can be translated to times on the global time line. In addition to the translated times, 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 experiments, it is nonintrusive to the system. For more detail, see [2].

2.4.2. Fault injection testing

Since Loki's notion of global state is imprecise in order to be less intrusive, it is possible that when faults are injected based on the global view of state, the injections will occur in the wrong state. The analysis tools are responsible for determining whether the fault injection(s) for an experiment occurred in the correct state(s). This requires that the user specify a *fault conditional*, which is a logical expression that can be used to determine the correctness of an injection. The analysis tools then determine the "truth value" of the fault conditional. For more detail, see [2].

2.4.3. Measure estimation

The final task of the analysis tools is to generate the appropriate measures for the fault injection campaign. Measures that might be desired by a user include the probability that a fault is covered, the time from fault occurrence to fault coverage, the probability of one of several different outcomes as specified by the user, and the time between the occurrence of two events. The results should include parameters of the time and probability distribution. Along with these details, confidence intervals for the measures should be computed.

In order to compute these measures, Loki must provide a measure description language. Using this language, the user must specify, for each fault conditional, the system state that indicates the coverage (or noncoverage) of a fault. Additionally, the user must specify which measure(s) are desired for that fault conditional.

2.5. Conducting a Fault Injection Campaign using Loki

Given this architecture, there are three steps to conducting a fault injection campaign in Loki. First, the user must specify information about the system and the campaign to be conducted, including physical descriptions of the events, the node state machines, the fault conditional, and the measures desired. For more information about these specifications, see Section 3.3. Second, the experiment(s) designed in step one must be executed using the Loki run time in conjunction with the system components. For details on how campaign execution is controlled and executed, see Section 3.4. Last, Loki's analysis tools must operate on the observations taken during the campaign in order to determine the requested results. For specifics about the interface between the run time and the analysis tools, see Section 3.7. For information about the operation of the analysis tools, see [1]. For a detailed example of an application instrumented to use Loki as an empirical evaluation tool, see Chapter 4.

3. RUN-TIME SOFTWARE ARCHITECTURE

3.1. Overview

This chapter discusses in more detail the implementation of the run time of the Loki empirical evaluation tool. Section 3.2 gives a description of the software architecture at the system level. Section 3.3 covers the state machine engine, which tracks the state of the system under study according to the user-defined state machines. Section 3.4 presents more detail about the state machine transport, which serves as the means of communication between state machines. Section 3.5 explains the recording of experiment observations, and Section 3.6 details how experiments are managed.

3.2. System

This section introduces the major pieces of the Loki architecture and their interactions. These components will be discussed in more detail in the succeeding sections.

3.2.1. Requirements

The Loki run time must meet the following requirements:

- The Loki run time must provide communicating state machines in order to track the partial global view of state. The user must be able to design the state machines to match a given system and fault injection campaign.
- The Loki run time must provide a fixed interface in order for the system under study to insert events into the state machines tracking the system state.
- The Loki run time must allow multiple points for fault injection and observation in a single system node.
- The Loki run time must efficiently collect and record system observations sufficient to generate the measures required by the user.

3.2.2. Software architecture

As described in Chapter 2 and illustrated in Figure 1, any system being validated with Loki consists of multiple nodes. Each node is instrumented with a piece of the Loki run time that tracks the partial view of system state for that node and sends messages to other nodes when necessary. In addition to the pieces of Loki instrumenting the nodes, there are global components of Loki necessary for facilitating communication between the nodes and for controlling experiments and fault injection campaigns.

The state machines in the instrumented nodes communicate via a *state machine transport,* which deals with the details of determining what a notification's destination hosts are and actually sends and receives notification messages. The state machine transport may or may not use the same network as the system under study. To prevent interference with packets on the actual network that are sent by the system under study, the user may elect to utilize an isolated network for communication between state machines.

Nodes in the fault injection campaign are named, and these *node names* are used by the state machines as destinations for notifications. Node names need not be unique: nodes that share a node name are considered part of a *node group,* and notifications sent by a state machine to a particular node name are forwarded to the entire node group. The sending of a notification to a single member of a node group is treated as a reliable point-to-point message; there is no sense of ordering or synchrony implied by this group structure. More detail about node groups and the means of communication between them is discussed in Sections 3.3 and 3.4.

The detailed software architecture for a node instrumented using Loki is shown in Figure 2. When using Loki to evaluate a system, each node involved in a fault injection campaign creates a single *state machine engine,* which will track the state of that node. Once this state machine engine exists, the node creates one or more probes (Section 2.3.3) to pass relevant events to the engine.

When created, the state machine engine creates a *recorder* object and, if it is a communicating state machine, a *local state machine transport* interface. The recorder records time observations for use by the analysis tools and is discussed in detail in Section 3.5. The local state machine transport provides the local portion of the functionality of the state machine transport (the communication mechanism for state machines). The local state machine transport is discussed in Section 3.4.

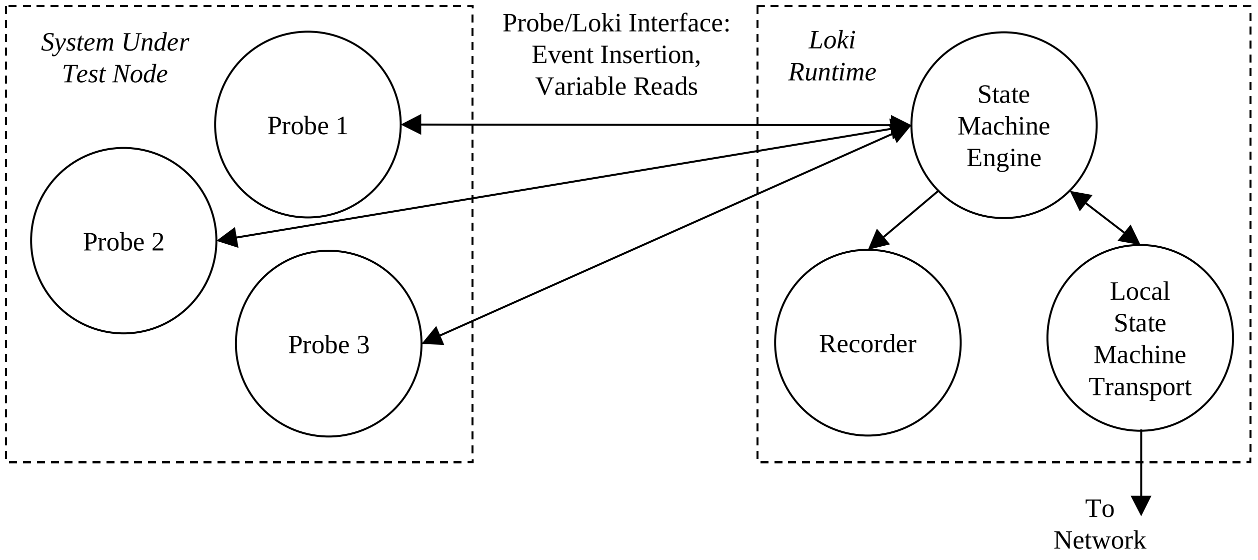


Figure 2: Local Node Software Architecture

Figure 3 shows several nodes involved in an experiment. Each node communicates with others via the state machine transport and takes observations using the recorder. There are two node groups, one containing three nodes and one containing only a single node. The treatment of these groups is the same in the implementation even though one of the groups is a singleton group.

The state machine transport shown in Figure 3 is made up of the local state machine transports contained in each local node along with a *global state machine transport*. The global state machine transport runs as a separate process from the system under study. There is a single global state machine transport for a fault injection campaign, and it runs separately from all system nodes. The global state machine transport, along with its relationship to the local state machine transport, is covered in detail in Section 3.4.

To run experiments automatically, a fault injection campaign must also have a single *global campaign controller*. The global campaign controller is a state machine responsible for starting and ending experiments using information gathered from the system nodes. This controller is discussed in Section 3.6.

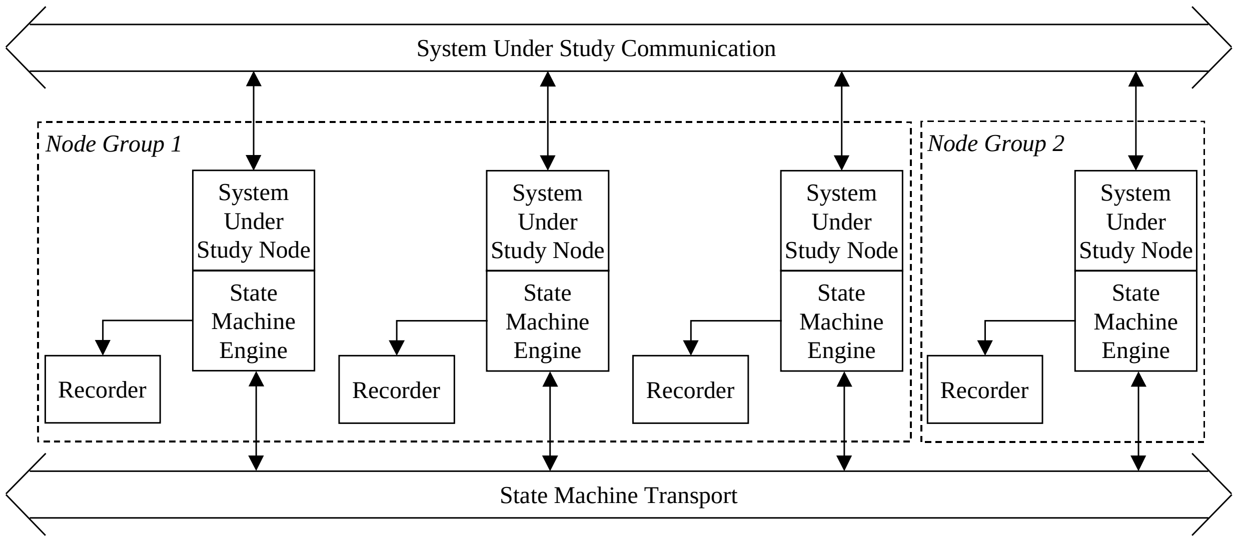


Figure 3: Nodes in an Experiment Using Loki

3.3. State Machine Engine

The state machine engine is responsible for keeping track of the state of the node with which it is associated according to the state machine specification supplied by the user.

3.3.1. Requirements

The state machine engine must meet the following requirements:

- State machines must provide a standardized interface for inserting events, getting feedback from an event insertion, and accessing the current state and state/state machine variables. This allows users to write instrumentation code in the way that best suits their system without needing any changes in the implementation of Loki.
- State machines must provide user-defined variables at the level of the state and at the level of the state machine. The probes should have access to the state machine variables. This allows the user to create a notion of "sub-state" that can be changed by local or global events but does not require the overhead of maintenance in the state machine engine.
- Communicating state machines must accurately track the state of the local node, but they may be (slightly) imprecise with respect to remote nodes. If communicating state machines do not accurately track the local state, then the actual state in which faults are

injected must be checked even at the local level. This added level of complexity would make it very difficult to design useful experiments.

- Communicating state machines must allow the associated node to have multiple probes. This allows for more observability in the system, which is required for many fault injection experiments.
- A noncommunicating version of the state machine engine must be provided for experiments that do not require a partial global view of state. This noncommunicating version will be of lower latency and therefore will be less intrusive to the system.

3.3.2. Software architecture

There are two types of state machine engines: communicating and noncommunicating. Communicating state machine engines can track a partial global state of the system. Noncommunicating state machine engines cannot track partial global state, but they provide a lower-intrusion solution for fault injection campaigns that do not require this knowledge. The core of both types of state machine engine has a fixed implementation; however, the actual definitions of the states, transitions, and actions must be described by the user. First, the user specifies a system-wide *event dictionary*, which assigns names to important system events and defines the means of recognizing them. Next, the user specifies a *state machine specification* for each type of node involved in an experiment. The state machine specification file describes the states in the state machine and the responses to be executed when a given event is received in a given state, including transitions between the states. These specification files are then translated into code by a state machine parser and an event dictionary parser implemented specifically for the Loki system. This generated code completes the state machine engine's implementation for a given experiment. This process is shown in Figure 4.

State machine engine core and campaign-specific functionality

Table 1 shows the important data members of the state machine engine class. The local state machine transport, used for communication with other nodes, will be discussed in Section 3.4. The recorder, used for data collection, will be discussed in Section 3.5. The rest of the fields are used for maintaining the state of the local node and reacting to system events as specified in the state machine specification.

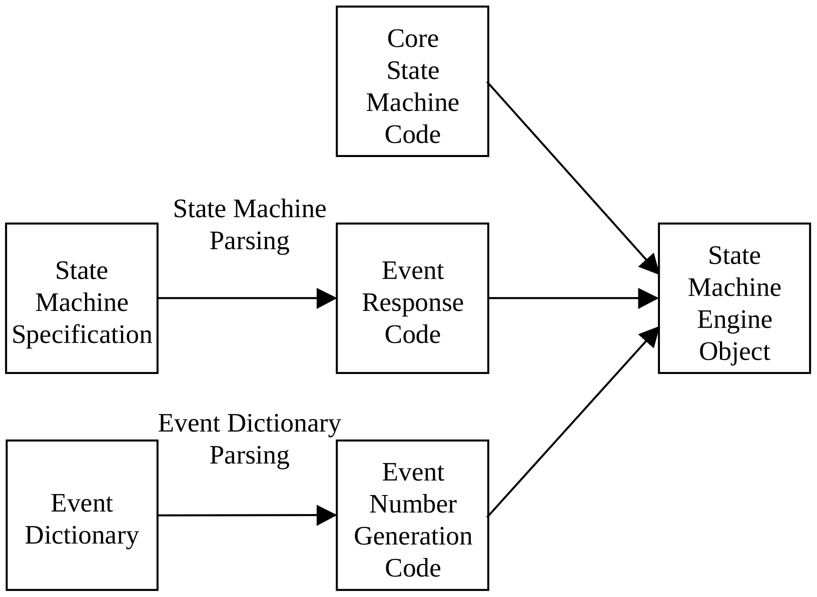


Figure 4: State Machine Object Creation from Existing Code and Specification Files

Table 1: State Machine Engine Data Members

Data Member	**Functionality**
Local state machine transport	Means of communication with other state machines
Recorder	Means to record system observations
Current state number	Number corresponding to the current state in the state machine
Event response table	References to functions to be executed in response to a new system event
Event response index table	Indexes for obtaining the correct event response from the event response table

States and events are represented by numbers. The state machine engine keeps track of the *current state number* (N_s) for use in observations and in determining how to respond to an incoming event. The *event response table* holds references to the functions to be used in responding to a system event. These functions are generated from the state machine specification. The *event response index table* is used when an event arrives to look up the index for the appropriate function in the event response table. The event response table has one entry for each state/event combination, but it is made up of short integers rather than integers. If there is no response for a state/event combination, the entry in the table will be a negative value. The event response table, which must contain integers, contains only as many entries as there are transitions (including self-transitions) in the state machine.

When a new event is received from a system probe, its event number (N_e) is determined using an event-matching function generated from the event dictionary. The index into the event response table ($N_s * N_e$) is then calculated. Using this value, the appropriate index for the event response function, if any, is taken from the event response index table, and the corresponding event response function is executed.

Event dictionary

The event dictionary delineates handles for the events relevant to a given fault injection campaign. Associated with each handle is a C-style Boolean expression that indicates how to recognize that event. Figure 5 illustrates this format. There should be one event dictionary for the entire system. For an example of an event dictionary, see Chapter 4 and Appendix A.

```
event_handle_1          Boolean expression 1
event_handle_2          Boolean expression 2
                  .
                  .
                  .
event_handle_N          Boolean expression N
```

Figure 5: Event Dictionary Format

The event dictionary parser generates the event matching function from this specification format. In the current implementation, the function is a series of `if` clauses, with the conditional for the `if` being the Boolean expression specified in the file and the return value being a number representing the event handle specified in the file. The first match causes the function to return.

State machine specification

The *state machine specification* describes the actual state machine that will track the node's state during the campaign. This specification lays out the states in the state machine and any responses to system events, including transitions between states. In addition, the user can specify both *state variables* and *state machine variables,* which are separate from the notion of the current state maintained by the state machine engine. State variables are used and seen only by the state with which they are associated. These variables allow the user to create a notion of *substate* within a state, if desired. The value of these variables is not examined or modified by the state machine engine in any way, unless they are accessed or changed by a user-written action function. Additionally, state variables are not accessible by probes. State machine variables, however, are accessible to all states as well as all probes. Both state and state machine variables are strictly for the use of the user-defined action functions and, in the case of state machine variables, the user-written probes. The state machine engine will neither access nor modify these variables.

State machine responses to events include transitions, variable updates, and notifications. In addition, the responses can include other user-defined code, such as a coin-flip to determine what the next state should be. These responses are specified as C/C++ code, with special structures for indicating notifications, variables, transitions, and return codes. The current implementation requires that users understand these special structures and incorporate them into the response code.

Figure 6 shows the format for the specification language. The first item in the specification is an optional list of any header files that are required for system calls or user-defined functions used in the file. Next is the list of state machine variables, which is also optional. Once these two lists are defined, the user must define one or more states. Inside of each state the user should list the events to which that state must respond. The response functions are defined as a combination of C/C++ statements and Loki-specific statements. The Loki-specific statements are used to probe return codes, notifications (if any), and next-state transitions (if any). For examples of the use of the specification language, see Chapter 4 and Appendix A.

Communicating state machines

Before introducing the probe interface, it is necessary to provide more detail regarding the two types of state machine engines, communicating and noncommunicating.

Communicating state machines must accept messages not only from the node probes but also from other state machines. Messages from other state machines arrive via the state machine transport (Section 3.4). To keep the global state more precise, the state machine engine should handle messages sent by other state machines as soon as possible. With a single thread of control, this cannot be done. If the state machine only checks for new events when a probe event is inserted (single noninterruptible thread of control in the node), arriving notification events will wait an arbitrary amount of time for processing. This makes the precision of global state dependent on the frequency of events in the node, making accurate fault injection more difficult.

```
include_list
      <header_file_1>
            ⋮
      <header_file_n>
end_include_list
var_list
      <state_machine_variable_1>
                  ⋮
      <state_machine_variable_n>
end_var_list
state <state_name_1>
      var_list
            <state_variable_1>
                  ⋮
            <state_variable_n>
      end_var_list
      event <event_name_1>
      <c_statement_1>
            ⋮
      <c_statement_n>
      <loki_statement_1>
            ⋮
      <loki_statement_n>
end_state
         ⋮
state <state_name_n>
         ⋮
end_state
end_sm
```

Figure 6: State Machine Specification Format

Instead, the communicating state machine engine can be accessed by two different thread types (Figure 7). One type, the application thread, executes application code and inserts events from the node into the state machine. There may be one or more application threads associated with a state machine. The second type, the transport thread, controls a portion of the local state machine transport (Section 3.4), receiving incoming notifications and passing them to the state machine engine as soon as they arrive. There is exactly one transport thread associated with each state machine.

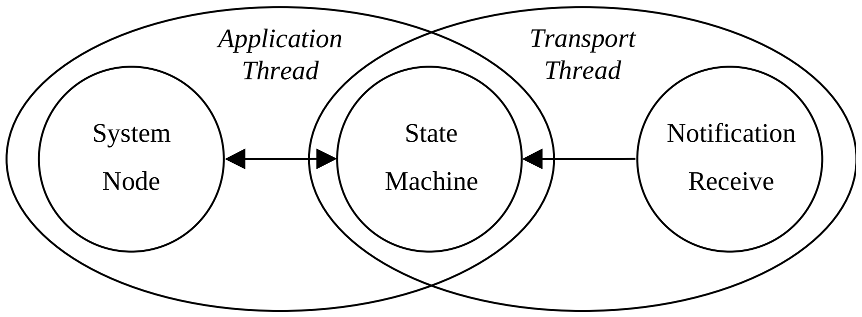


Figure 7: Threads in Communicating State Machines

There is a restriction created by this architecture: all accesses to the state machine must be atomic. If accesses are not atomic, it is possible for a thread to observe partial modifications to the state machine or for threads to be in a race condition to modify the state machine. To meet this restriction, mutual exclusion of accesses to the state machine is guaranteed by a lock. Because access to the state machine is guaranteed to be atomic by a lock, it is possible to allow access and event insertion to different probes in different threads or processes of a node. This meets the restrictions laid out for this architecture.

Noncommunicating state machines

The noncommunicating state machine engine does not need this level of synchronization, since no notifications are received on the network. On initialization, the noncommunicating state machine engine must register with the global state machine transport (Section 3.4) in order to receive instance and experiment numbers; however, after doing this the engine does not need to access the transport manager again until leaving the experiment.

The lack of synchronization on the state machine does place restrictions on the use of the noncommunicating architecture. Specifically, there cannot be probes in more than one process or thread in a node. All probes in a node with a noncommunicating state machine must be in the same process and thread. If this were not the case, nonatomic accesses to the state machine might occur, inducing race conditions that could cause incomplete reads of variables or incorrect settings of the state of the system. This restriction is in contrast to that of the communicating state machine architecture described above. An exception to the restriction on the noncommunicating state machine architecture occurs if the user provides his or her own synchronization on the noncommunicating state machine.

Probe interface

As described in Section 2.3.3, Loki provides a standard interface to the state machine engine through the probe. Using this probe interface, the user can design instrumentation that can insert events, get feedback from the state machine, and read state machine variables. This interface must be via method calls rather than a message-passing means of communication in order to satisfy the requirement that state changes caused by local node events be tracked accurately. If a message-passing interface such as a socket is used, uncertainty in the state is introduced because the message is not necessarily consumed by the state machine immediately after it is produced by the probe. If the event generates a state change, the state machine engine may not see that state change immediately after it happens, due to this delay time. When a method call is used, and the method call does not return until any state machine transitions induced by the event have been completed, additional latency is incurred; however, the requirement regarding accurate local state is satisfied because the application cannot generate another state change before the current one is processed.

The Loki run time provides a *basic probe* class, which can be used to instrument user code in one of three ways. First, the user may use the probe as-is, treating it as an interface and adding extra instrumentation to execute fault injections or other actions based on the feedback from the state machine outside of the probe. Second, the user can inherit from the probe class, enhancing the basic probe methods as desired and adding additional data members and methods as necessary to create a system-specific probe class. Finally, the user can combine these two approaches to instrument the system under study.

Table 2 specifies the methods that are defined in the basic probe. These methods fall into three categories. First, there are event insertion methods. These methods allow the probe to pass an event to the state machine, possibly modifying the system state. In the communicating state machine architecture, these methods will cause a lock to be acquired on the state machine engine to prevent synchronicity violations. If they induce the recording of an observation in the state machine engine, they will also cause a lock to be acquired on the recorder. The current implementation supports only a single kind of event insertion method, which passes the data immediately on to the state machine. Second, there are user record functions. These methods allow the probe to record some event of interest, such as a fault injection. In the communicating state machine architecture, these methods will cause a lock to be acquired on the recorder but not on the state machine. This allows some parallel access with the state machine transport. Third, there are methods for accessing the state machine variables and state. In the communicating state machine architecture, these methods must cause a lock to be acquired on the recorder for atomicity of state machine variable reads and modifications. The current implementation supports two state machine access functions, one for reading the state machine variables and one for reading the current state of the state machine.

Table 2: Probe Interface Methods

Probe Interface Method	Functionality
`Insert(void * msg)`	Insert a message into the state machine engine.
`Record(char data)`	Generate an observation time stamp and an observation.
`Record(unsigned long t_high, unsigned long t_low, char data)`	Generate an observation using a known time stamp.
`void * GetStateMachineVariables()`	Read the user-defined state machine variables.
`unsigned char GetCurrentStateNum()`	Read the current state number.

3.4. State Machine Transport

The *state machine transport* provides the mechanism for communication between state machines associated with different system nodes.

3.4.1. Requirements

The state machine transport must meet the following requirements:

- State machines must be able to send notifications to a node group according to a *node name* for that group. All nodes within a node group share the same name. This allows the user the flexibility to design experiments based either on specific nodes being in specific states or on specific types of nodes being in specific states.

- The implementation must ensure that all notifications in the system can reach their destination node group without being routed through a single fixed process and/or host. This prevents a bottleneck in the communication between state machines.
- Nodes must be able to dynamically enter and exit an experiment, allowing broader experiment design.
- Each node within a node group, and therefore sharing a node name, must be assigned an *instance number* that is unique within a single experiment in a fault injection campaign. Instance numbers are not currently used in the run time, although they should eventually be used to make the sending of notifications to nodes more specific. Instead, instance numbers are used in the determination of the correctness of a fault injection [1].

3.4.2. Software architecture

When a node starts, it knows only its own node name and its own host information. In order to send notifications to other nodes, it must be provided with more information. The state machine transport is the means of obtaining this information. The transport is divided into the *global state machine transport manager* and the *local state machine transport manager* (Figure 8). Every node in an experiment has its own local state machine transport; however, there is only one global state machine transport for a fault injection campaign.

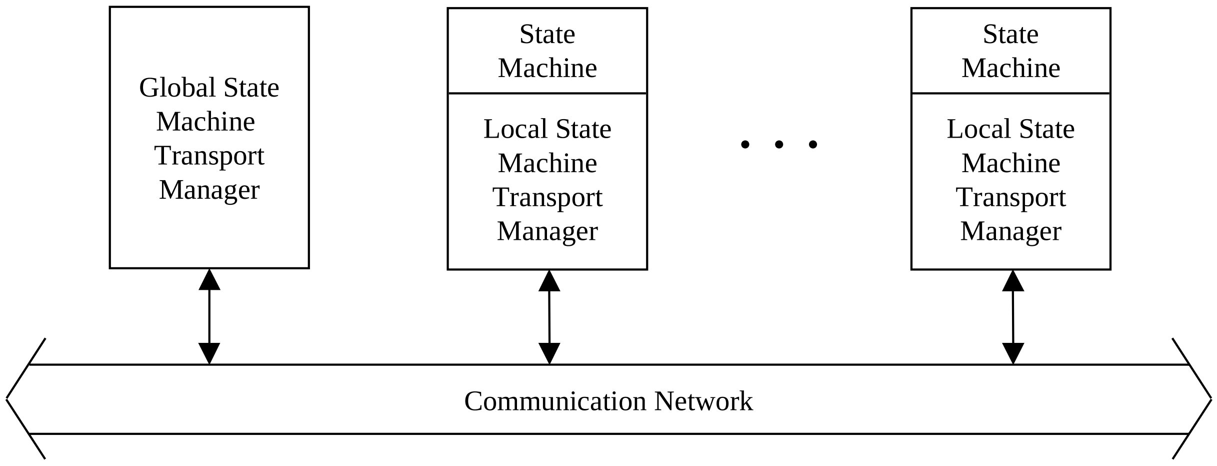

Global State Machine Transport Manager
State Machine
Local State Machine Transport Manager
• • •
State Machine
Local State Machine Transport Manager
Communication Network

Figure 8: Global and Local State Machine Transport Managers

The global state machine transport manager acts as a server for the membership of the experiment at any given time. This allows nodes to dynamically enter and exit an experiment. When a new node joins an experiment, the global transport manager transmits the membership list to the new node. In addition, the global transport manager assigns to each new node an *instance number* that provides it with a unique identifier within its node group. Finally, the global transport manager also keeps a count of the number of experiments that have been performed so far and transmits the experiment number to the new node. The global transport manager resets the membership list and increments the experiment number when it receives a reset request from the global campaign controller, described in Section 3.6.

The local state machine transport manager is responsible for taking the membership list it receives from the global transport manager and requesting socket connections with all of the nodes on the list. After that is done, the local transport manager opens a listening socket and accepts connections with any new nodes joining the experiment. In addition, when the state machine engine (Section 3.3) sends a notification to a certain node group, the local transport manager must forward it to the appropriate destination(s). When a node exits the experiment, its local transport manager is responsible for informing the global transport manager and all other local transport managers. Finally, when new information is received on the connections to other local transport managers during an experiment, the local transport manager must determine whether it is a notification or an exit request and act accordingly.

Experiment membership and message ordering

Given these algorithms for determining experiment membership and sending messages to node groups, it is important to note two key concepts about the node group facility provided by Loki that have the potential to cause problems in an experiment. The first concept is that membership lists are imprecise, and the second is that no global message ordering is provided. Although algorithms exist in group communication systems (such as Ensemble [11]) for maintaining a consistent membership view as well as global ordering of messages, these algorithms are not desirable in a system such as Loki since these algorithms have overhead that would cause Loki to be more intrusive to the system.

When designing an experiment using Loki, it is important to remember that the notion of membership is imprecise, which means that for a given node *n* at a given time *t* it is possible that another node *m* has joined the experiment but is not yet known by *n*. That means that whenever node *n* makes a decision based on its notion of the membership of a group or of the experiment as a whole, such as sending a message to all members of a particular group, it is possible that that decision is incorrect. That incorrectness is a factor of allowing nodes to join and exit dynamically rather than as a result of providing the notion of groups. Even without the notion of group structure, instrumentation code or state machine responses might still make decisions based on which nodes are currently taking part in an experiment, and those decisions still have the potential to be incorrect.

There are two cases in which experiment membership is known to resolve completely. One case occurs when no more members join or leave an experiment. In this case, after all nodes have formed or terminated all connections, the membership list can be relied on to be correct. The second case is more limited. When a certain subset of experiment membership stops changing, then the membership view of that set of nodes becomes stable. This means that decisions made by the state machine responses or user instrumentation based on this subset of the membership will be correct. In cases aside from these two, decisions are correct depending on whether the group has been stable "long enough" for the node making the decision to have a correct view of the membership; however, some experiments might have outcomes that are not affected by a decision that is incorrect.

The second issue affecting experiment design is that there is no concept of global message ordering provided by Loki. Providing this type of message ordering requires too much overhead and would therefore significantly slow down the exchange of notifications between state machines. This has a significant impact on experiment design. If it is possible that notification messages could be received in more than one order with the same end result, the user should design the state machines to provide multiple paths, one corresponding to each possible notification message ordering. Using knowledge of the system under study and exchanging notifications only when absolutely necessary for the measure-driven partial global view of state should keep the state machines from becoming unmanageably large.

Node table

Membership lists in both the global transport manager and the local transport manager are maintained using a *node table.* The node table uses a hash table for node group name lookup and an array for storing information about the node to be used in operations involving all nodes. Both types of transport manager can insert and delete nodes in the node table one at a time. The local transport manager can also insert lists of nodes, so that when it first receives a membership list, it only needs to call a single insertion function. In addition to this, local transport managers can request that the node table set up connections with the nodes currently in the table. This functionality is used after a node first receives the membership list. The node table also provides functionality specific to the global transport manager. This functionality allows the global transport manager to request that the host table write the membership list array to a socket, and it is used whenever the global transport manager needs to send the membership list to a node joining the experiment.

The information that is stored about each node in the *node information structure* is described in Table 3. The node information structure contains actual data rather than pointers to data. This allows node information structures to be passed between the global and local transport managers without first copying them into an appropriately sized buffer. Unfortunately, this forces the system to limit the length of node names so that they can be stored in a statically rather than dynamically sized array.

Table 3: Fields in the Node Information Structure

Data Field	Usage
Node name	Stores the name of the group to which the node belongs
Host address	Stores information about the internet address of the node's host and the port number to be used for contacting it for connections
Socket	Stores the socket used in communicating with this node
Index	Indicates the index where this node's information is stored in the node table's linear array
Instance number	Indicates this node's global instance number
Number of instances	In the hash table's dummy list header, indicates the number of instances seen so far in this experiment
Next	Indicates the next node information structure in the list for a node group

The first three fields of the node information structure are necessary for forming connections and sending notifications between nodes. The node name indicates the group of which the node is a member. All nodes within the same group share a node name, and the destination for a notification is specified as a node group name. The socket address and socket number are used for the actual communication with a given node. The next field, the instance number, is used by the analysis tools, and is kept in the node information structure for the purpose of recording.

The last two fields are for use in the hash table, which is used by the node table for looking up node groups. Since node names are not unique, the hash table stores a list of all nodes with a given node name (i.e., a node group) and indexes that group by the node name. When a request is made to the hash table to look up a particular name, and no additional information is specified, the hash table returns the entire list. This is useful when a node wishes to send a message to an entire node group. To look up a specific node within a group, the hash table requires that the caller specify the socket number that provides the means for communication between the caller's process and the requested node.

Keeping the entire group as a list indexed by the node name also allows the hash table to assign each node an instance number, which provides the node with a unique identifier within its node group. This is done by keeping a "dummy" list head that is used specifically for counting the number of instances that have been seen in this experiment. This current instance number count is not reset until the node table, through the global transport manager, receives a reset request from the global campaign controller (Section 3.6).

For implementation reasons, the socket number is used instead of the instance number to look up a specific node. Since the socket number is already stored in the state machine transport, and determining the instance number would require looking in the node information structure stored in the node table, it is faster and more space-efficient to use the socket number for these operations.

State machine transport classes

Both the local state machine transport and the global state machine transport are derived from a single state machine transport class. This class contains an array whose elements are the sockets connecting the transport manager to other nodes; an array whose elements correspond to those sockets; management data for those arrays and for determining when the sockets have incoming information; and a single node information structure containing the node information for this transport manager**.**

Each type of state machine transport contains a host table with the appropriate functionality for that transport manager. The host table is an actual objects rather than a pointer because accesses to the host table will be frequent and should not incur the extra overhead of dereferencing a pointer. Because of this, the host table must be in the derived class rather than the base class.

In addition to its specific host table, the global transport manager contains a pointer to a list of *incomplete connections*. This list is needed because a node may exit the experiment immediately after the membership list is transmitted to a new node but before the new node has connected to it. When this happens, the new node will not know that the exiting node no longer exists and will therefore continue to try to connect to it. To prevent this, every node must inform the global transport manager when it is done making all necessary connections. Until a node does this, it is considered an incomplete connection, and all exit notifications will be forwarded to it.

Figure 9 shows an example of an order of execution that requires that the global manager maintain the list of incomplete connections. In this example, the leftmost node (Node 0) is joining the experiment, and the other nodes (Node 1, Node 2, and Node 3) are already in the experiment. First, Node 0 tells the global transport manager that it is joining the experiment. In return, the global transport manager adds Node 0 to the list of incomplete connections and transmits the membership list to it. The membership list at this time contains Node 1, Node 2, and Node 3. Once Node 0 receives the membership list, it attempts to set up connections with every node on the list. While Node 0 is in the process of setting up these connections, but before it connects to Node 3, Node 3 exits the experiment, informing the global transport manager that it is leaving and closing all of its existing connections. Since Node 0 had not yet connected to Node 3, Node 0 is not informed of this exit by Node 3; instead, the global transport manager forwards the information to its list of incomplete connections, including Node 0.

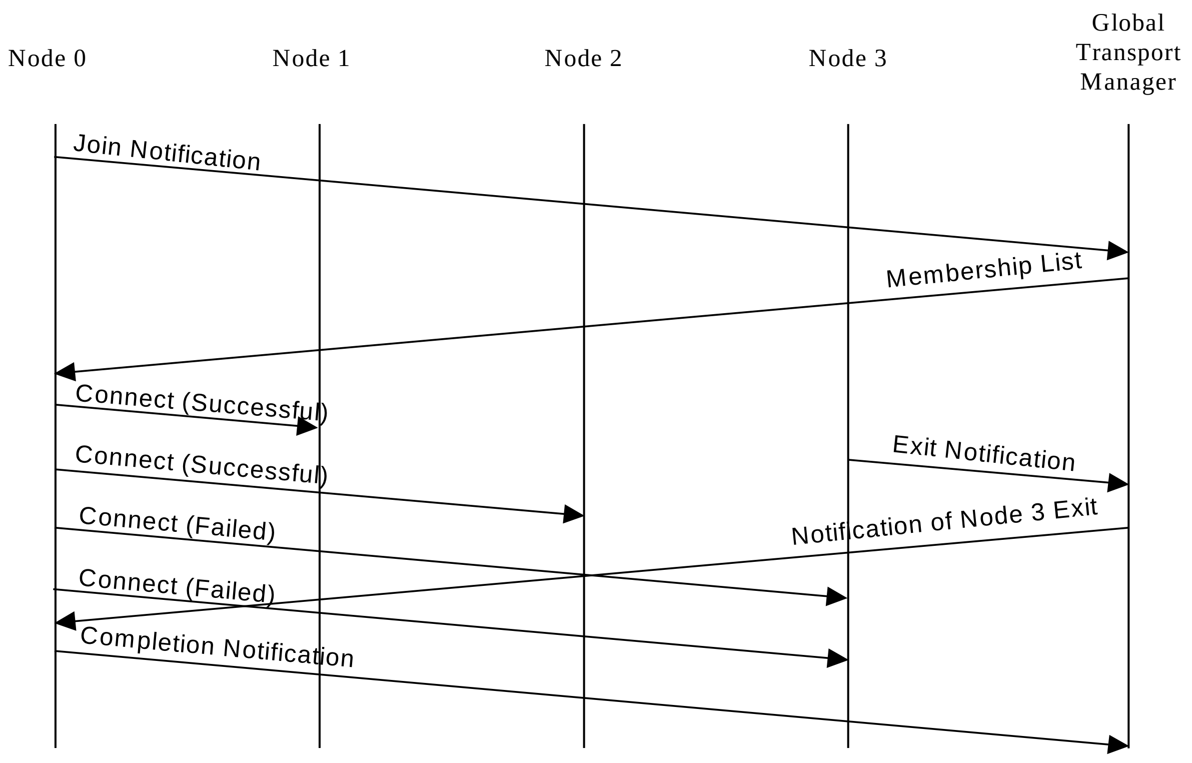


Figure 9: An Example Use of the List of Incomplete Connections

From the perspective of Node 0, the connection attempts with Node 1 and Node 2 are successful, but the connection to Node 3 fails on the first attempt. Node 0 continues to attempt connection setup since it is possible that Node 3 is still in a setup phase itself and is therefore not ready to accept connections. After receiving a second connection failure, Node 0 receives the global transport manager's message indicating that Node 3 is no longer in the experiment. At this time, Node 0 discards Node 3 from its copy of the membership list and ceases the attempt to connect with Node 3.

Dynamically joining an experiment

To deal with nodes joining the experiment, the global state machine transport waits for requests for the membership list from new nodes and then transmits the list to the node making the request. To do this, the global transport manager maintains a TCP listening socket. When a connection request is made, the global transport manager assumes that it comes from a new node and sends the membership list after first receiving the information for the new node. Only one such request is handled at a time, preventing an instance in which two nodes might receive the same membership list and each never know about the existence of the other. The new node is placed on the incomplete connection list until it informs the global transport manager that it is done with initialization procedures. If any other node informs the global transport manager that it is closing while the new node is still on the incomplete connection list, the new node is informed that that node is closing. This algorithm is shown in Figure 10.

From the new node's perspective, when it joins an experiment, its local state machine transport sends its node information to the global transport manager and obtains the experiment membership list from the global state machine transport. Once it has this list, the new node's transport manager inserts the nodes in the list into its node table. Using the information provided, which includes the node's address and the port on which it may be contacted to request a connection, the new node contacts each node in the list, establishing a connection and transmitting its own information to that node. After these connections are made, the node has successfully joined the experiment, and notifications can be sent directly to their target node(s) via the connections. At this time, the new node sends a connection complete notification to the global transport manager. This algorithm is shown in Figure 11.

After a node has joined an experiment, its local transport manager becomes responsible for granting connections from new nodes joining the experiment. This is done by maintaining a listening socket. When connection requests are received on this socket, the node reads the information for the node requesting the connection and adds that node to its node table. This algorithm is shown in Figure 12.

Operation during an experiment

Once connections to other nodes in the experiment have been made, notifications can be sent directly to the other nodes. To send a notification, the state machine transport decodes the destination list and looks each destination node group name up in the node table. The message is then written to each socket in the list for each node group to receive the message.

```
$\forall$ join message received from node i
    transmit membership list and experiment information to node i
    membership list $\leftarrow$ membership list $\cup$ {i}
    list of incomplete connections $\leftarrow$ list of incomplete connections $\cup$ {i}
$\forall$ completed connection message received from i
    list of incomplete connections $\leftarrow$ list of incomplete connections – {i}
$\forall$ exit message received from i
    membership list $\leftarrow$ membership list – {i}
    if i is on the list of incomplete connections
        list of incomplete connections $\leftarrow$ list of incomplete connections - i
    $\forall$ node n on the list of incomplete connections
        forward message notification to n
    close connection with i
$\forall$ reset command received from the global campaign controller
    $\forall$ node n on the membership list
        transmit abort command to n
    membership list $\leftarrow$ $\phi$
    list of incomplete connections $\leftarrow$ $\phi$
```

Figure 10: Global Transport Manager Operation

```
transmit join message to global transport manager
receive membership list and experiment information from global transport manager
$\forall$ nodes n in the membership list
    repeat
        attempt connection to n
        $\forall$ node j exit message received from G
                membership list $\leftarrow$ membership list – {j}
        $\forall$ abort command  received from G
                terminate
    until
        connected to i or membership list no longer contains i
transmit connection complete message to G
```

Figure 11: Local Node Initialization

```
∀ connection request from node i that is accepted
    membership list ← membership list ∪ {i}
∀ notification message with destination g received from the state machine
    ∀ nodes i in group g
        transmit notification to node i
∀ notification message received from node i
    insert notification message in the state machine
∀ exit message received from node i
    membership list ← membership list – {i}
    close connection to i
∀ abort command received from the global transport manager
    terminate
```

Figure 12: Local Transport Manager Operation During an Experiment

The local transport manager must also process incoming messages. The manager uses the `select()` function to determine when a message is received. The `select()` function returns whenever one of the connections to the other nodes or to the global transport manager has an incoming message on it. When a local transport manager receives a notification message from another node, it simply passes it on to the state machine via the appropriate method call. In addition to notifications from other state machines, the local transport manager may also receive an abort message from the global state machine manager. If this happens, the local transport manager must immediately terminate its node. These algorithms are shown in Figure 12.

Dynamically exiting an experiment

When a node exits an experiment, its local transport manager first informs the global transport manager that it is leaving the experiment, so that it will not appear on the membership list sent out to new nodes. The local transport manager then informs all other nodes that it is leaving the experiment and closes its connections with those nodes. When a remote node’s local transport manager receives such a notification, it removes the node from its membership list and closes its end of the network connection. Each of the remaining local nodes that has finished initialization needs to remove the exiting nodes from its node table and close their end of the network connection. Nodes that are still in the process of initialization (as described in the previous section) must remove the exiting node from the node table. This way, if the node has not yet been connected to, the initializing node will not attempt to connect to it. These algorithms are shown in Figures 12 and 13.

```
transmit exit message to global transport manager
close connection to global transport manager
∀ nodes i in the membership list
    transmit exit message to i
    close connection to i
exit
```

Figure 13: Local Node Termination

When the global transport manager receives such an exit request, it first forwards the exiting node's information to any incomplete connections. It then removes the node from its node table and from the list of connected sockets, closing the socket in the process. This operation is atomic from the perspective of requests to the transport so that an incorrect membership list is not sent out to any new nodes. This algorithm is detailed in Figure 10.

Experiment reset

Unlike other local nodes, the fault injection campaign controller (Section 3.6) can send an experiment reset command to the global transport manager. When the global transport manager receives a reset request from the controller, it first informs all local nodes of the termination so that they may exit. It then requests that the node table reset. To do this, the node table removes all of the nodes in its membership list except for the campaign controller (Section 3.6.2). Doing this causes the instance numbers to reset in the hash table. In addition, the global transport manager increments the experiment number. This algorithm is shown in Figure 10.

3.5. Recorder

The recorder provides the data collection facilities of the Loki run time. Observations taken by the recorder during the experiment include both changes in the state of the system and user-specified events such as fault injections. It is important for these data collection methods to be as accurate as possible to get the best results.

3.5.1. Requirements

The recorder must meet the following requirements:

- Data collection mechanisms must allow observations to be made from either the probes/user instrumentation or the state machine engine. This allows the user to make observations immediately after injecting a fault.
- Data recording should be low latency in order to minimize intrusion on the system.
- Observation time stamps should be taken as close to an event as possible to get results that are as accurate as possible.

3.5.2. Software architecture

Results collected on-line are saved into a file for use after the campaign by the analysis tools. The object responsible for this process is the *recorder* object. As discussed in Section 3.3, the recorder is a member of the state machine engine and, in the communicating state machine architecture, is protected by a lock in order to provide atomic accesses. The state machine may access the recorder directly; however, the user probes must make a call via the state machine engine in order to have access to the locks. To prevent overhead from these extra function calls, the methods involved are implemented as inline functions.

Observations and time stamps

In the course of experiment execution, all state changes are recorded since they may be needed by the analysis tools, e.g., for fault injection testing [2]. In addition to these observations, the user may specify observations that may represent, for example, fault injections or events of interest for measurement.

Time stamps are generated using an instruction specific to Pentium®-compatible chips and newer x86-compatible chips. This instruction (`rdtsc`) allows direct access to system counters, unlike the Unix-compatible `gettimeofday()` function. Using this instruction allows lower-latency access to time stamps. For more detail, see [2].

Time stamps for an event should be read as close to the event as possible; to this end, a means for first generating and then recording time stamps is provided as an inline function in the Loki run time. This allows times to be taken before the lock on the recorder is acquired, which is advantageous since acquiring a lock is a high-latency operation. The user can call this function in the probe or can manually obtain the time stamps using a macro provided by the Loki system that is a wrapper for the `rdtsc` instruction. The user would then call a different `Record()` function in the probe interface, passing the times as well as the observation data as arguments and then recording them. (See Section 3.3.2 for details on the probe interface.)

Recorder architecture

The recorder object takes observations and writes them into a file. Experiment observations are passed to the recorder as a time stamp, which consists of two long integers, the current state value, a character integer (i.e., a byte on this system), and a character representing observation information. If a character is N bits long, it can represent 2^N different values, each of which might represent a different type of event in the system. For example, the user might be injecting three correlated faults. In that case, the user would define three different values representing those faults and record those values as observation information when the corresponding fault is injected. Two of these values for observation information are reserved by the Loki system. One of these values is recorded as the observation information with a state change, and the other is recorded as the observation information when an *exit event*, indicating that a node has left the experiment, is recorded.

Direct file writes can be slow, depending on the system and the system load. To lower this latency, the observation file is *memory-mapped*. Memory-mapping is a facility provided by Unix-style operating systems that allows portions of a file to be mapped to an array that appears as part of the program memory. When the end of the portion currently mapped is reached, the next portion is mapped, so the file is still extensible. In that way, the program sees writes as writes to an array and therefore physical memory rather than as writes to a file on the disk.

Each separate node has its own output file for an experiment. The output files are named according to the node name and the instance number. This makes them easier to "find" after the fault injection campaign is complete. At the beginning of an experiment, the recorder attempts to open the file in which it is to record its observations. If the file does not exist, the first thing the recorder does is create the file and output the node name and instance number into it. Then, whether the file did or did not exist, the recorder outputs a header delimiter and outputs the header information for this experiment-i.e., the host name, the experiment number, and the instance number. Once this is done, the recorder is ready to take any observations for this experiment. When recording an observation, the recorder first writes an observation delimiter and then writes the information passed to it by the call to its `Record()` method.

3.6. Fault Injection Campaign Controller

Loki requires a fault injection campaign controller to start experiments, determine when experiments have ended, and determine when the appropriate number of experiments (a constant number determined by the user) has been run.

3.6.1. Requirements

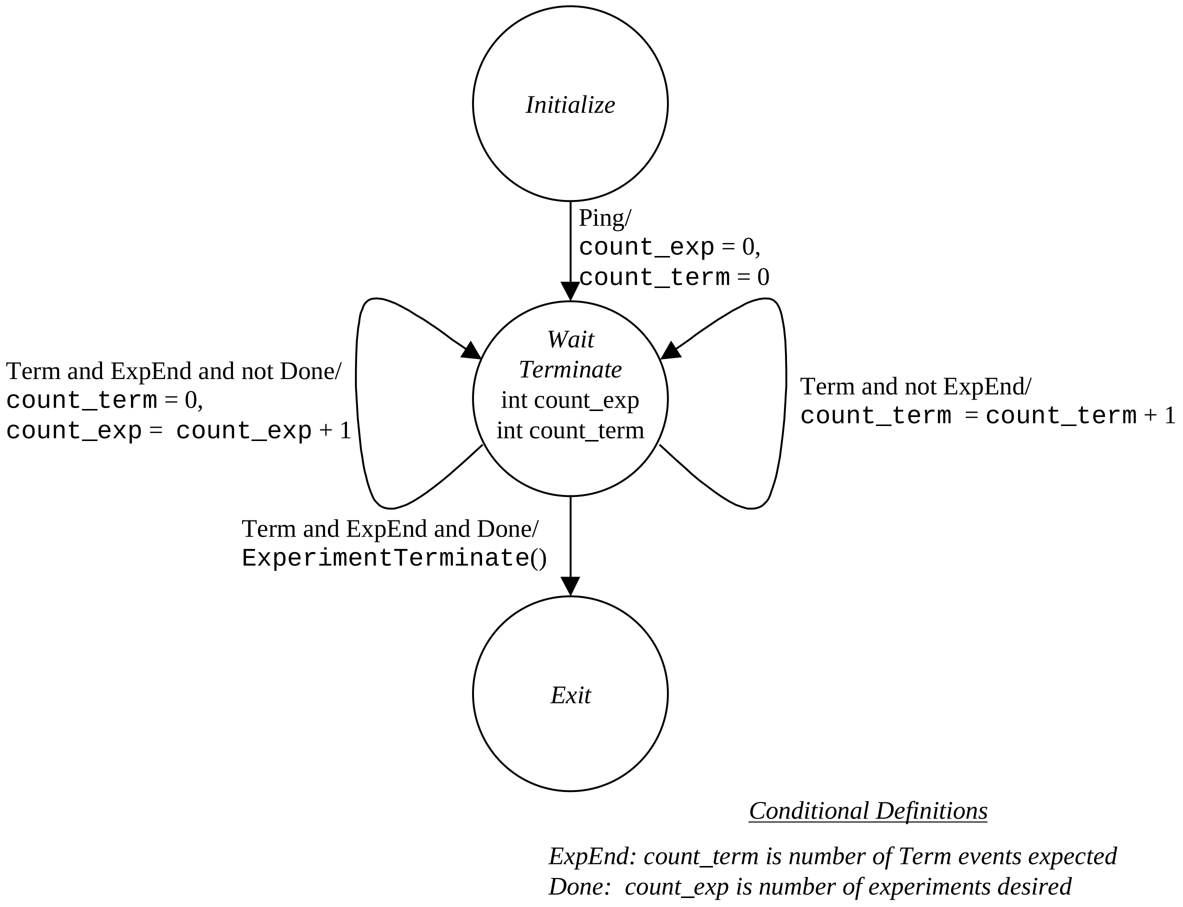
The fault injection campaign controller must meet the following requirements:

- It must be able to start experiments by running a fixed list of executables on certain hosts.
- It should be able to determine when experiments have ended using a non-campaign-specific and non-system-specific interface.
- It should be able to determine when the appropriate number of experiments has been run.

3.6.2. Software architecture

In order to meet these requirements, the campaign controller must be a member of the experiment. To be a member of the experiment, the campaign controller is implemented as a simple application instrumented with a communicating state machine. The only function of the campaign controller's application is to create the state machine engine and a single probe. Once this is done, the application inserts a single event into the state machine via the probe and immediately calls `sigsuspend()` so that it will be as nonintrusive to the system as possible.

The state machine, via the state machine transport thread, does the majority of the work of the controller. The three states in the campaign controller state machine (Figure 14) are *Initialize*, *Wait Terminate*, and *Exit*[1]. The state machine begins in the *Initialize* state, and when the application inserts its event (PING), the state machine transitions to *Wait Terminate*. On this transition, the state machine executes a script that starts the initial nodes in the system.



Initialize
Ping/
count_exp = 0,
count_term = 0
Wait Terminate
int count_exp
int count_term
Term and ExpEnd and not Done/
count_term = 0,
count_exp = count_exp + 1
Term and not ExpEnd/
count_term = count_term + 1
Term and ExpEnd and Done/
ExperimentTerminate()
Exit
Conditional Definitions
ExpEnd: count_term is number of Term events expected
Done: count_exp is number of experiments desired

Figure 14: Global Campaign Controller State Machine

Once it is in the *Wait Terminate* state, the state machine simply waits for termination events (TERM) that are delivered to it by one or more nodes in the experiment. The state machine counts these termination events using a state machine variable, `count_term`, and when it reaches a user-defined number of these events, it considers the experiment over. At that time, the state machine increments another state machine variable, `count_exp`, which

[1] The *Exit* state is conceptual only. It does not need to appear in the state machine specification because execution ends at this point.

keeps track of how many experiments have been completed. If this value is equal to the number of experiments the user wished to have performed, the state machine terminates the campaign. Otherwise, it sends a reset signal to the global state machine transport (Section 3.4.2), causing the global state machine transport to remove all nodes from the node table, to reset the instance numbers in the node table, and to increment its value for experiment number. After the reset has been sent, the controller sleeps for a mandatory amount of time specified by the user prior to system compilation. This suspension time between experiments allows application system resources such as sockets to be reset, and as such must be set by a user familiar with the system under study.

Since the campaign controller can only determine when to terminate an experiment based on receiving a fixed number of termination events, it does not allow very flexible determination of the end of an experiment. If the user needs more flexibility, he or she can write an additional controlling state machine with more system- or campaign-specific intelligence. This additional controller can then be responsible for sending a single termination event to the fault injection campaign controller.

In addition to this restriction, the current architecture of the tool only allows campaigns to include a single type of experiment. This is because new state machines and/or new probes and instrumentation code must be written for a new experiment, and then the system executables must be recompiled. To conduct a true fault injection campaign consisting of multiple types of experiments, the user must conduct multiple Loki campaigns.

3.7. Interfacing Between the Run Time and the Analysis Tools

To simplify the design and implementation of the analysis tools, the Loki run time provides a facility for extracting experimental results from the output files generated by the recorder. This facility, called the *experiment data manager*, understands the format in which data is recorded during a fault injection campaign and can read it into a known set of data structures for use by the analysis tools.

To analyze the data from a given experiment in a fault injection campaign, the analysis tools request the data for that experiment from the data manager. The data manager determines which output files contain data from the experiment and creates from this data *observation lists* for each host, which are sorted according to the time of occurrence of each observation. The observation lists separated according to hosts have a known ordering, since all observations on each list were recorded with time stamps taken from the same system clock. From these *deterministic observation lists*, the analysis tools can create a *composed observation list*, which contains data from one or more hosts and therefore might have an ambiguous ordering. Operations to test for the proper injection of faults and to determine the value of experimental measures are performed on the composed observation list of data from all hosts participating in an experiment.

4. TESTING AND VALIDATION

4.1. Overview

This chapter discusses the testing of Loki using two different test applications. Section 4.2 discusses the requirements for applications designed to be used in the testing of Loki. Sections 4.3 and 0 cover the two test applications and the results obtained. Specifically, Section 4.3 discusses an application designed primarily for functional verification, and Section 0 details an application developed for performance analysis.

4.2. Requirements

There are two different goals in validating a system such as Loki. First, there is functional verification. In the case of Loki, the need for functional verification leads to the following requirements for a test application:

- Use at least three different state machines. One must be driven by local events, one by notifications, and one by a combination of the two. This allows functional errors in the responses to events to be filtered to the relevant portion of the code.
- Use at least two different notification types, to assist in the verification of the event-response functionality.
- Define at least two different types of nonnotification system events, also to assist in the verification of the event-response functionality.
- Test node group sizes of one, two, and greater than two. This allows the testing of the functionality of the state machine transport and the node table.
- Run multiple experiments using the campaign controller, with varying numbers of termination events.
- Take observations independent of the state machine in order to test the functionality of the recorder.

The second goal is to determine performance results for Loki. Specifically, the time to transmit a notification from one state machine to another should be determined under different transmission frequencies and node group sizes. This leads to the following requirements for a test application:

- Allow determination of the time to transmit a notification.
- Allow changing of the frequency at which notifications are sent.
- Allow scaling of the sizes of both the group sending the message and the group receiving the message.

To satisfy these two different sets of goals, two different test applications have been developed. The first, the pyramid application, is used for functional verification and is discussed in Section 4.3. The second, the send/receive application, is used for performance analysis and is discussed in Section 0. Included with the discussion of the send/receive application are performance output results (Section 4.4.2).

4.3. Pyramid Test Application

The main focus of the pyramid test application is functional verification of Loki. Functional verification for Loki includes ensuring that the state machine transport, state machine engine, recorder, and campaign controller all function according to their specifications.

4.3.1. Application architecture

The pyramid application is a very simple hierarchical leader election protocol. There are three levels of hierarchy in the application. These are the leaf level, the leaf manager level, and the manager level. At the manager level, there is always a single process that controls the entire arbitration scheme. In the instance of the pyramid application used to test Loki, there are two leaf manager processes that report to the manager, and each leaf manager process has three leaf processes reporting to it, for a total of six leaf processes. Figure 15 illustrates this architecture.

In this protocol, each leaf group[1] elects a leader within its group. To do this, each process in the group chooses a number, and the processes then communicate those numbers to each other. A process knows it is the leader if its number is the highest one. Ties are resolved

[1] Note that these groups do not necessarily correspond to Loki node groups.

by repeating this arbitration between tied members until a single process remains. Once the leader process has been determined, that leader process must forward that information along with the “winning” number to the group’s leaf manager process. The leaf manager processes then determine which leaf group’s leader should be the global leader, by comparing the leaf leader numbers in a fashion similar to the leaf group arbitration. Finally, the elected leader of the leaf managers must communicate that it is the leader and send the election number to the manager process.

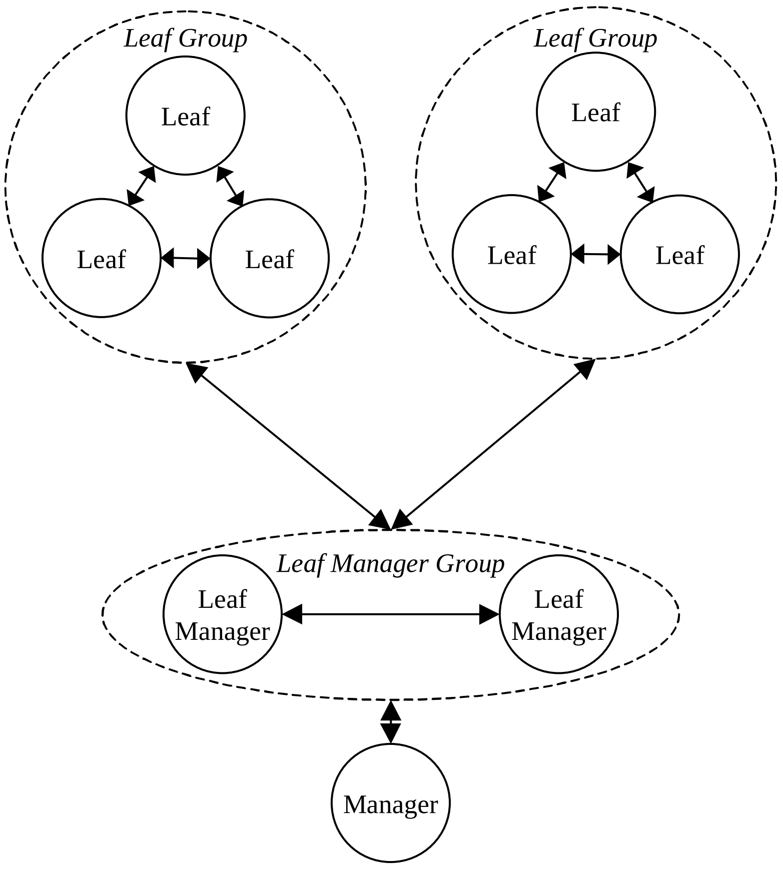


Figure 15: Pyramid Application Architecture

4.3.2. Validation architecture

The pyramid application instrumented using Loki has three node groups. These are the Leaf group, corresponding to all leaf processes; the LeafManager group, corresponding to all leaf manager processes; and the Manager group, corresponding to the single manager process. Events in the pyramid application are shown in Table 4, and Table 5 specifies which events are used as notifications and between which groups they are exchanged. For the actual event dictionary as well as the state machine specification files used for instrumentation of the pyramid application, see Appendix B.

Table 4: Events in Pyramid Application

Event Name	Occurs in Group	System or Notification	Indicates
INIT_DONE	Leaf	System	Leaf has completed socket setup to other leaf processes.
	Manager	System	Manager has completed socket setup.
FOLLOWER	Leaf	System	Leaf has completed election and is not the leader.
LEADER	Leaf	System	Leaf has complete election and is the group leader.
CRASH	Leaf	Notification	Manager has told the leaf that is the group leader to crash.
LEAF_ELECT	LeafManager	Notification	Leaf state machine has informed leaf manager that it is in the process of leader election.
LEAF_DONE	LeafManager	Notification	Leaf state machine has informed leaf manager that it is done with election process.
WAIT_DONE	Manager	System	Manager has received information from leaf manager and is done waiting.
EXIT	Leaf	System	Application has exited.
	LeafManager	System	Application has exited.
	Manager	System	Application has exited.

Table 5: State Machine Communication in Pyramid Application

Notification Event	Sending Group	Receiving Group
LEAF_ELECT	Leaf	LeafManager
CRASH	Manager	Leaf
TERM	Leaf	Controller
	LeafManager	Controller
	Manager	Controller

The state machine for the Leaf node group, shown in Figure 16, contains states for initialization (*Init*) and election (*Elect)* that must be traversed by all leaf nodes. After the *Elect* state, nodes that are elected as leaders traverse the *Lead* state and the *Crash* state. Nodes that are not elected as leaders go to the *Follow* state. Upon application termination, both types of node go to the *Exit*[1] state and terminate. All transitions are caused by local events with the exception of the transition to the *Crash* state. Notifications sent by the Leaf group are LEAF_ELECT, sent on the transition from *Init* to *Elect*, and LEAF_DONE, sent on the transition from *Elect* to *Lead*. Both of these notifications are intended to trigger state changes in the LeafManager state machine.

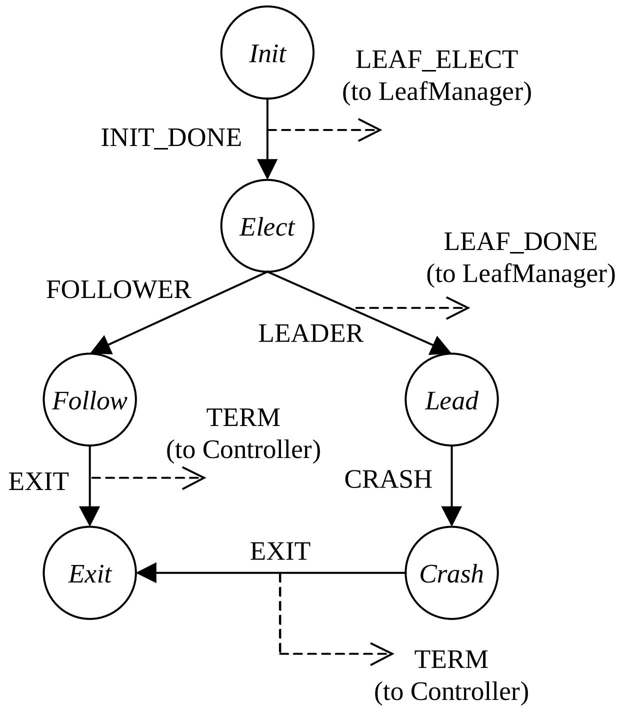


Figure 16: Leaf Node Group State Machine

The LeafManager node group state machine, shown in Figure 17, is driven only by notifications sent by the Leaf state machine. The states traversed by a leaf manager process are *Leaf Init*, *Leaf Elect*, and *Done*. In *Leaf Init*, the leaf manager process waits for a notification that the leaf processes are done with initialization. In *Leaf Elect*, the leaf manager process waits for a notification that the leaf group has elected a leader. In *Done*, the leaf manager process waits for the leaf manager application to terminate.

The Manager node group state machine, shown in Figure 18, is driven only by local events. The states traversed by this process are *Init*, *Wait*, and *Done*. The process remains in the *Init* state while the manager sets up sockets to receive from the leaf manager. After this is done, a local event causes the state machine to transition to the *Wait* state, where it remains until the leaf manager contacts the manager process at the application level. At that time, the

[1] As in Section 3.6, the *Exit* state is conceptual only.

state machine transitions to the *Done* state, sending a CRASH notification to the leaf group and then waiting for the application to terminate.

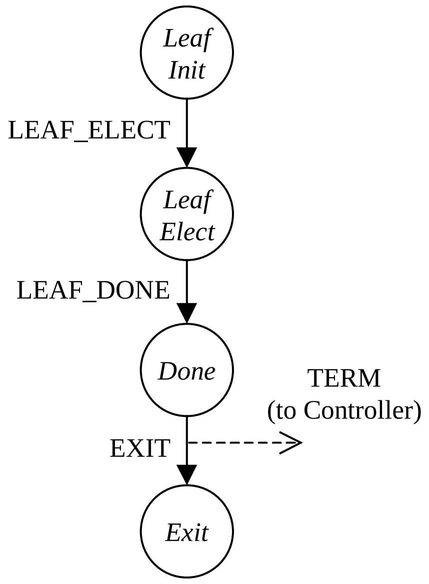


Figure 17: LeafManager Node Group State Machine

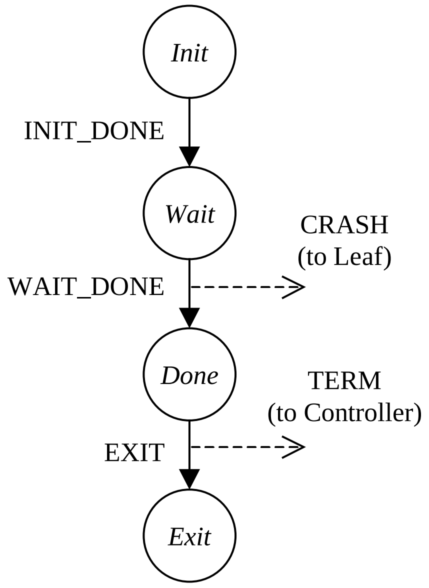


Figure 18: Manager Node Group State Machine

4.4. Send/Receive Application

The major purpose of the send/receive application is to provide performance metrics for Loki. The instrumentation for this architecture consists of two state machines, with one sending notifications and the other receiving them.

4.4.1. Application and validation architecture

The send/receive application consists of two types of processes. One, the sender process, calls `sleep()` a given number of times and inserts an event into the state machine each time it "wakes up." The other, the receiver, simply suspends after creating the Loki state machine. The events used in this system are described in Table 6.

There are two groups in the send/receive application: the Sender group and the Receiver group. The groups correspond to the process types of the same name. There is only one type of experiment-specific notification used. That event is the PING_NOTIFY, which is sent from the Sender group to the Receiver group. The Sender group state machine (Figure 19) is made up of a *Send* state and an *Exit* state. A sender process remains in the *Send* state until application termination. There is a single self-transition in the *Send* state, which is caused by a local PING event. When this self-transition occurs, a notification (PING_NOTIFY) is sent to the Receiver group.

Table 6: Events in Send/Receive Application

Event Name	Occurs in Group	System or Notification	Indicates
PING	Sender	System	State machine should send notification to receiver group.
PING_NOTIFY	Receiver	Notification	Sender has sent a notification.
EXIT	Sender	System	Application has terminated.
	Receiver	System	Application has terminated.

The Receiver group state machine (Figure 20) is made up of a *Recv* state and an *Exit* state. The receive process also remains in the *Recv* state until application termination. There is a single self-transition in this state, which is triggered by receipt of the PING_NOTIFY event sent by the Sender group state machine.

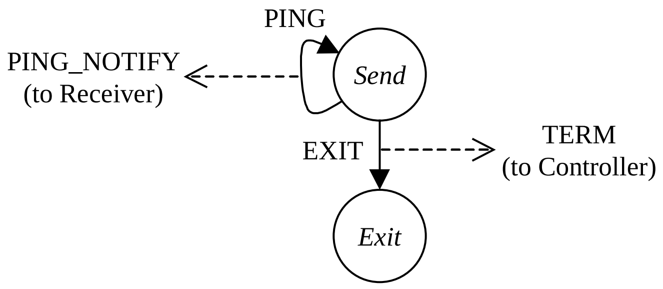


Figure 19: Sender Node Group State Machine

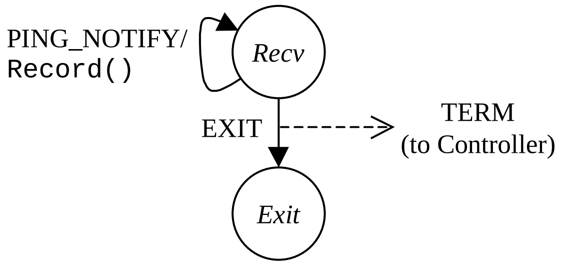


Figure 20: Receiver Node Group State Machine

4.4.2. Results

In this section, results are presented from the use of the send/receive application. These results focus on the time to transmit a notification from one state machine to another. In each type of experiment, notifications are sent by the state machines of the sender group at a given frequency until a given number of notifications has been sent. The size of the Sender group, the size of the Receiver group, and the frequency of notifications vary across experiments, with the baseline experiment having a Sender group size of one, a Receiver group size of one, and a notification frequency of one notification per second. Each type of experiment was repeated 20 times to obtain the results presented here. The various experiment types performed are described in Table 7.

Figure 21 shows the value for the transmission time of each notification observation averaged over all runs of the baseline experiment. It can be seen from the figure that only transmission times near the beginning of the experiment take on large values. In the steady state of the experiment, the values are much lower and vary much less.

Table 7: Experiments Performed Using the Send/Receive Application

Sender Group Size	Receiver Group Size	Notification Frequency (messages/s)	Observations per Experiment
1	1	1	50
1	1	10	50
1	1	20	50
1	1	50	50
1	1	100	50
1	2	1	50
1	5	1	50
1	10	1	50
2	1	1	50
5	1	1	50
10	1	1	25

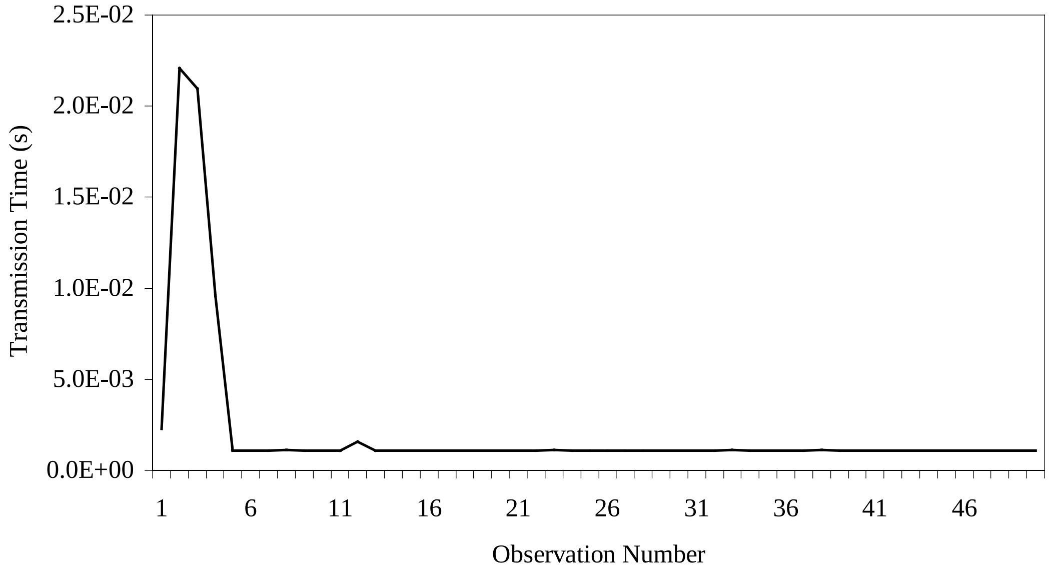


Figure 21: Notification Transmission Times in Baseline Experiment

Figure 22 shows the same information for experiments using one sender and one receiver with notification transmission frequencies of 10, 20, and 100 per second. The same trend seen in Figure 21 holds; however, it is important to note that the steady-state value of transmission times for the experiment with notification transmission frequency of 100 notifications per second is higher than that for the other experiment types.

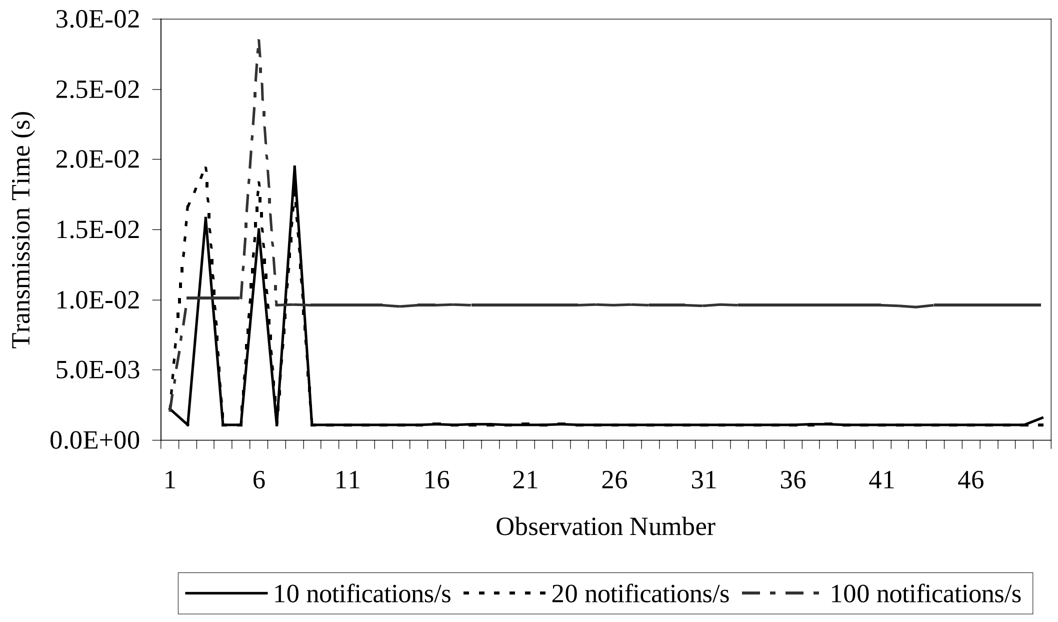


Figure 22: Notification Transmission Times under Varying Notification Frequency

Table 8 presents the mean transmission times for each experiment, both over all observations and excluding the first 10 transmission times in each experiment. It can be seen from these results that the mean steady-state transmission times are lower than the overall mean transmission times. Additionally, some trends can still be seen in the mean steady-state transmission times. The mean time to transmit a notification increases as the Sender and Receiver group sizes increase. In addition, the mean time to transmit a notification increases for a notification frequency greater than 20 messages per second.

Table 8: Mean Notification Transmission Times

Sender Group Size	Receiver Group Size	Notification Frequency (messages/sec)	Mean Transmission Time (ms)	Mean Steady State Transmission Time (ms)
1	1	1	2.1	1.1
1	1	10	2.1	1.1
1	1	20	2.5	1.1
1	1	50	6.1	5.8
1	1	100	9.9	9.6
1	2	1	2.2	1.4
1	5	1	3.8	2.9
1	10	1	5.1	4.9
2	1	1	2.0	1.2
5	1	1	1.4	1.3
10	1	1	4.3	1.3

5. CONCLUSION

The Loki empirical evaluation tool, developed as part of the work presented in this thesis, is intended to assist in the evaluation of software distributed systems through fault injection and measurement capabilities. The tool is designed to determine statistically significant measures relating to fault injection as well as to allow injection of faults based on the state of multiple components of the system under study. The work covered in this thesis describes the means for allowing the injection of faults based on a partial global view of the state of the system.

To accomplish this goal, the run-time experiment framework allows the user to define state machine descriptions of the operation of a node. The user indicates via this state machine specification the relevant states that can be visited by a node, the system events that cause transitions between the states, and any additional actions that are taken in response to an event. Included in the specification of these event-response actions are notifications that are used to exchange relevant state information between state machines. Nodes are partitioned into groups, and notifications are sent to an entire group. Using a well-defined interface, the user can write a probe and instrumentation code that will insert faults based on the current state of the state machine. The probe can determine when to inject faults either actively, by polling the state of the state machine regularly, or passively, by using signals and/or return codes. During the experiment, observations are taken to be used by the analysis tools in the generation of the desired measures, and a global controller ensures that the correct number of experiments (as specified by the user) is executed.

The Loki run time has been tested using two primary test applications. One application is intended for functional testing and the other for performance analysis. Results generated from the performance test application indicate that, in general, mean notification transmission times are under 10 ms, and that in the steady state of an experiment, notification transmission times are much lower.

Future work for the Loki run-time experiment framework includes further performance enhancements to the core of the Loki run time, further performance analysis of the Loki run time, and enhancements to the usability of the Loki empirical evaluation tool.

Performance enhancements should be determined and implemented, using the results of the performance analysis guide to determine which functional portions of the run time can be most improved and what performance enhancements would be most helpful in designing and executing useful experiments. As an example, consider the results discussed in Section 4.4.2. While the notification times at the beginning of an experiment were high, after an initial transient, all notification times were low. Before making any modifications to improve the transmission time of notifications early in the experiment, the developer should consider how this will affect the transmission times in the experiment's steady state, and whether this effect will make the design of useful experiments more difficult.

In the area of usability improvement, a user interface for the entire Loki tool would be desirable. This interface would allow a user to define state machines, associate state machines with particular nodes, design experiments/campaigns including different node/state machine pairs, and specify fault conditionals and desired measures for an experiment/campaign. The interface should generate the input files discussed in Chapter 3. Additionally, the interface could perform global operations on the state machines in an experiment prior to experiment execution (for example, to determine if the state machines might send notifications too frequently or could possibly deadlock). Work on this interface should be done based on the work presented in this thesis in conjunction with that presented in [2].

APPENDIX A: TEST APPLICATION SPECIFICATION FILES

This appendix contains the actual specification files used in the scenarios described in Chapter 4. Figures 23, 24, 25, and 26 show files used by the pyramid application. Respectively, they show the event dictionary and the state machine specifications for the LeafManager, Leaf, and Manager groups. Figures 27, 28, and 29 show files used by the send/receive application. Respectively, they show the event dictionary and the state machine specifications for the Receiver and Sender groups. Figure 30 shows a state machine specification for a campaign controller (Loki Manager).

```
EXIT         current_message->contents[0] == 3
INIT_DONE    current_message->contents[0] == 4
LEADER       current_message->contents[0] == 5
FOLLOWER     current_message->contents[0] == 6
CRASH        current_message->contents[0] == 7
LEAF_ELECT   current_message->contents[0] == 8
LEAF_DONE    current_message->contents[0] == 9
WAIT_DONE    current_message->contents[0] == 10
```

Figure 23: Pyramid Application Event Dictionary

```
state LEAF_INIT
      event LEAF_ELECT
                 tfr.next_state = LOKI__LEAF_ELECT;
                 return FALSE;
      end_event
end_state
state LEAF_ELECT
      event LEAF_DONE
                 tfr.next_state = LOKI__DONE;
                 return FALSE;
      end_event
end_state
state DONE
      event EXIT
                 tfr.host_num[0] = 0; // loki_manager
                 ((char *)(tfr.data))[0] = LOKI_MANAGER_TERM_NOT_VAL;
                 tfr.n_dests = 1;
                 tfr.next_state = -1;
                 return TRUE;
       end_event
end_state
end_sm
```

Figure 24: LeafManager Group State Machine Specification

```
state INIT
      event INIT_DONE
                 tfr.next_state = LOKI__ELECT;
                 tfr.host_num[0] = LEAF_MANAGER_NODE_GROUP;
                 ((char *)(tfr.data))[0] = PYR_ARB_NOTIFY_VAL;
                 tfr.n_dests = 1;
                 return TRUE;
      end_event
end_state
state ELECT
      event LEADER
                 tfr.next_state = LOKI__LEAD;
                 tfr.host_num[0] = LEAF_MANAGER_NODE_GROUP;
                 ((char *)(tfr.data))[0] = PYR_ARB_DONE_NOTIFY_VAL;
                 tfr.n_dests = 1;
                 return TRUE;
      end_event
      event FOLLOWER
                 tfr.next_state = LOKI__FOLLOW;
      end_event
end_state
state LEAD
      event CRASH
                 sme->Record((char) 40, (bool) TRUE); // Crash event
                 tfr.next_state = LOKI__CRASH;
                 return FALSE;
      end_event
end_state
state FOLLOW
      event EXIT
                 tfr.host_num[0] = 0; // loki_manager
                 ((char *)(tfr.data))[0] = LOKI_MANAGER_TERM_NOT_VAL;
                 tfr.n_dests = 1;
                 tfr.next_state = -1;
                 return TRUE;
       end_event
end_state
state CRASH
      event EXIT
                 tfr.host_num[0] = 0; // loki_manager
                 ((char *)(tfr.data))[0] = LOKI_MANAGER_TERM_NOT_VAL;
                 tfr.n_dests = 1;
                 tfr.next_state = -1;
                 return TRUE;
       end_event
end_state
end_sm
```

Figure 25: Leaf Group State Machine Specification

```
state INIT
      event INIT_DONE
                  tfr.next_state = LOKI__WAIT;
                  return FALSE;
      end_event
end_state
state WAIT
      event WAIT_DONE
                  tfr.next_state = LOKI__DONE_MANAGER;
                  tfr.host_num[0] = LEAF_NODE_GROUP;
                  memcpy(tfr.data, &c, sizeof(char));
                  ((char *)(tfr.data))[0] = PYR_CRASH_NOTIFY_VAL;
                  tfr.n_dests = 1;
                  return TRUE;
      end_event
end_state
state DONE_MANAGER
      event EXIT_EVENT
                  tfr.host_num[0] = 0; // loki_manager
                  ((char *)(tfr.data))[0] = LOKI_MANAGER_TERM_NOT_VAL;
                  tfr.n_dests = 1;
                  tfr.next_state = -1;
                  return TRUE;
       end_event
end_state
end_sm
```

Figure 26: Manager Group State Machine Specification

```
EXIT         current_message->contents[0] == 3
PING         current_message->contents[0] == 7
PING_NOTIFY  current_message->contents[0] == 8
```

Figure 27: Send/Receive Application Event Dictionary

```
state RECEIVE
      event PING_NOTIFY
                  sme->Record((char) 30, (bool) TRUE); // notify event
                  tfr.next_state = -1;
                  return FALSE;
      end_event
      event LOKI_EXIT_EVENT
                  tfr.n_dests = 1;
                  tfr.host_num[0] = 0; // loki_manager
                  tfr.next_state = -1;
                  ((char *)(tfr.data))[0] = LOKI_MANAGER_TERM_NOT_VAL;
                  return TRUE;
     end_event
end_state
end_sm
```

Figure 28: Receiver Group State Machine Specification

```
state WAIT
      event PING
                  tfr.next_state = -1;
                  tfr.n_dests = 1;
                  tfr.host_num[0] = RECEIVER_NODE_GROUP;
                  ((char *)(tfr.data))[0] = PING_FORWARD_VAL;
                  return TRUE;
      end_event
      event LOKI_EXIT_EVENT
                  tfr.n_dests = 1;
                  tfr.host_num[0] = 0; // loki_manager
                  tfr.next_state = -1;
                  ((char *)(tfr.data))[0] = LOKI_MANAGER_TERM_NOT_VAL;
                  return TRUE;
      end_event
end_state
end_sm
```

Figure 29: Sender Group State Machine Specification

```
var_list
             int count_term;
             int count_exp;
end_var_list
state LOKI_MANAGER_INITIALIZE
      event LOKI_MANAGER_PING
                 system("expmgr.pyramid.notimes\0");
                 ((LOKI__SMVARS *)sm_vars)->count_term = 0;
                 ((LOKI__SMVARS *)sm_vars)->count_exp = 0;
                 tfr.next_state = LOKI__LOKI_MANAGER_WAIT_TERMINATE;
                 return FALSE;
      end_event
end_state
state LOKI_MANAGER_WAIT_TERMINATE
      event LOKI_MANAGER_TERM_NOTIFY
                 cout << "Received LOKI_MANAGER_TERM_NOTIFY" << endl;
                 ((LOKI__SMVARS *)sm_vars)->count_term++;
                 cout << "count_exp = "
                      << ((LOKI__SMVARS *)sm_vars)->count_exp
                           << endl;
                 if (((LOKI__SMVARS *)sm_vars)->count_term == 9) {
                    cout << "Received all TERMs" << endl;
                    ((LOKI__SMVARS *)sm_vars)->count_exp++;
                    if (((LOKI__SMVARS *)sm_vars)->count_exp >= 5) {
                       cout << "All experiments complete; terminating." << endl;
                            ExperimentTerminate();
                    } else {
                       ExperimentReset();
                       system("expmgr.pyramid.notimes\0");
                       ((LOKI__SMVARS *)sm_vars)->count_term = 0;
                    }
                 }
      end_event
end_state
end_sm
```

Figure 30: Loki Manager State Machine

REFERENCES

[1] J. –C. Laprie, “Dependable computing: Concepts, limits, challenges,” in *Proceedings of the 25th International Symposium on Fault-Tolerant Computing, Special Issue,* 1995, pp. 42-54.

[2] D. Henke, “Loki – an empirical evaluation tool for distributed systems: The experiment analysis framework,” M.S. thesis, University of Illinois, Urbana, IL, 1998.

[3] W. G. Bouricius, W.C. Carter, and P.R. Schneider, “Reliability modeling techniques for self-repairing computer systems,” in *Proceedings of the 24th National Conference,* 1969, pp. 295-309.

[4] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell, “Fault injection and dependability evaluation of fault-tolerant systems,” *IEEE Transactions on Computers,* vol. 42, pp. 913-923, August 1993.

[5] G. Alvarez and F. Cristian, “A centralized failure injection environment for the validation of distributed fault-tolerant protocols,” University of California at San Diego, La Jolla, CA, Technical Report CS95-458, 1995.

[6] 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*, 1995, pp. 204-213.

[7] 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*, 1992, pp. 28-35.

[8] S. Dawson, F. Jahanian, and T. Mitton, “ORCHESTRA: A fault injection environment for distributed systems,” University of Michigan, Ann Arbor, MI, Technical Report CSE-TR-298-96, 1996.

[9] 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*, 1995, pp. 494-501.

[10] C. E. Ellingston and R. J. Kulpinski, “Dissemination of system time,” *IEEE Transactions on Communications*, vol. COM-21, pp. 605-623, May 1973.

[11] M. Hayden, “The ensemble system,” Ph.D. dissertation, Cornell University, Ithaca, NY, 1998.