© Copyright by Ryan Michael Lefever, 2003

AN EXPERIMENTAL EVALUATION OF THE CODA DISTRIBUTED FILE SYSTEM USING THE LOKI STATE-DRIVEN FAULT INJECTOR

BY

RYAN MICHAEL LEFEVER

B.S., University of Illinois at Urbana-Champaign, 1999

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, 2003

Urbana, Illinois

ABSTRACT

Experimental evaluation is an important way to assess distributed systems, and fault injection is the dominant technique in this area for the evaluation of a system's dependability. For distributed systems, network failure is an important fault model. Physical network failures often have far-reaching effects, giving rise to multiple correlated failures as seen by higher-level protocols. This thesis presents an experimental evaluation, using the Loki fault injector, that provides insight into the impact that correlated network partitions have on the Coda distributed file system. In the evaluation, Loki creates a network partition between two Coda file servers, during which updates are made at each server to the same replicated data volume. Upon repair of the partition, a client requests directory resolution to converge the diverging replicas. At various stages of the resolution, Loki causes a second correlated network partition, thus allowing the correlated partition's impact on the system's performance and availability to be evaluated. This thesis begins with the details of how Loki is used to evaluate a system. Then it presents a lightweight fault injection proxy as an approach to using Loki that is employed in the Coda evaluation. An overview of the Coda evaluation case study is provided. Finally, experimental results are presented.

For my Mother and Father, to whom i am forever grateful
for their unending love and unfaltering support

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor William H. Sanders, for his support in all aspects of my well being, and his dedication to his research group and to producing excellent research. I thank him for his research and academic guidance, and for the opportunity to be a member of his research group. Without him, I would not have gone on to graduate school. I would also like to thank Professor Michel Cukier (now of the University of Maryland) for his role as an additional advisor. He has always provided me with excellent insight.

In addition, I am extremely thankful and blessed to have had Ramesh Chandra and Kaustubh Joshi as research partners while working on Loki. They are two of the brightest people I know and have always challenged me to produce my best work. Luckily, they too were night owls and were always willing to have memorable discussions and to help me understand that which I did not. They are also very good friends, and they put up with me.

I am thankful to Jenny Applequist for editing my thesis and any other documents that I brought before her. She has tremendously improved my quality of writing. She is also a very important key to the success of Professor Sanders's research group.

I would like to thank the other members of the Perform research group with whom I have had the pleasure of working, namely Amy Christensen, Graham Clark, Tod Courtney, David Daly, Dan Deavours, Salem Derisavi, Jay Doyle, Vishu Gupta, Sudha Krishnamurthy, Vinh Lam, James Lyons, Prashant Pandey, HariGovind Ramasamy, Jennifer Ren, Paul Rubel, Sankalp Singh, Aaron Stillman, and Patrick Webster. They have all been integral to my success and to my enjoyment of my graduate career. They have shared numerous conversations with me, helped me whenever asked, given me friendship, and been willing to provide a welcome break from time to time. In particular, I thank James Lyons, who is my roommate and a great friend, and who also puts up with me.

I am thankful to the funding agencies that have financially supported my research. This material is based upon work supported by DARPA under grants F30602-96-C-0315, F30602-97-C-0276, and F30602-98-C-0187 and by the National Science Foundation under Grant No. CCR-00-86096. Any opinions, findings, and conclusions or recommendations expressed in this material are mine and do not necessarily reflect the views of DARPA or the National

Science Foundation.

I would like to thank my family–Michael, Joyce, Darin, and Shannan–and my friends for their support throughout my whole life. They make my life extremely enjoyable, and they put up with me the most. Finally, I would like to thank God for everything He has given me, which I cannot even begin to put into words.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

Distributed systems are becoming increasingly common, and are being used in many diverse environments, such as web servers, distributed multimedia, distributed operating systems, and mobile and ubiquitous computing environments. The requirements of these systems vary widely. For instance, a group communication system may have strict message ordering requirements, while a distributed multimedia system may have timeliness and throughput guarantees to meet. Some of their requirements can be reasoned about using formal methods. According to [1], formal methods are approaches in which "requirement specifications are developed and maintained using mathematically trackable languages and tools." However, formal proofs are susceptible to errors and may be based on inaccurate assumptions; furthermore, it is often infeasible to formally examine every aspect of a system. The implementation of a system can have a significant impact on correctness and performance as well. For instance, the assignment of thread priorities and timeout values, choice of communication properties, error and exception handling, and introduction of human errors will likely have serious ramifications for a system. Thus, it is preferable to validate certain requirements using experimental techniques. The motivation for using experimental validation could be the complexity of the system being validated, or the nature of the requirements considered. Furthermore, experimental techniques can be used to validate the implementation of a system, testing for design faults that may have been introduced in the implementation.

One technique that shows significant promise for distributed system validation is fault injection. *Faults* are potential causes of errors to a system, and *fault injection* is the process of experimentally evaluating a system with respect to those faults. Fault injection studies of network systems have been conducted for purposes such as computing coverage metrics and fault removal. For example, a study of an atomic multicast system [2] and of a Myrinet high-speed network [3] were conducted to compute coverage metrics, while [4] focused on

fault removal in studies of TCP and a group membership protocol.

This thesis reports on efforts to use fault injection to evaluate Coda [5], a distributed file system developed at Carnegie Mellon University. Coda is interesting because of its elaborate design for maintaining high availability, supporting mobile computing, and managing shared resources. Coda is designed to tolerate two fault models: network partitions and crash failures. Coda claims to achieve high availability by employing server replication and supporting disconnected operation under an optimistic consistency strategy. Due to the difficulty of experimentally evaluating a system as large and elaborate as Coda, this work focuses on one aspect of the system, namely the impact of correlated network partitions on Coda's data resolution techniques. Resolution is defined as "the process of converging diverging replicas to the same value" [6, p. 13].

A large body of research exists for Coda [5], including work on Coda's resolution techniques [6, 7, 8]. One team conducted an empirical study of Coda over a six-month period [9]. The study focused on the performance of Coda and frequency of failures from field data. To the best of our knowledge, there has not been a fault injection study of Coda or of any distributed file system.

1.1 Existing Fault Injectors

Several fault injectors exist for the study of networked systems, including EFA [10], DOCTOR [11], Orchestra [12], and NFTAPE [13]. These fault injectors represent many research advances. EFA was designed to test if a system adheres to the system's functional specification. It supports randomly generated fault cases, user-defined fault cases, and fault cases derived from analysis of source code, to inject communication faults. DOCTOR is a fault injection tool that was developed for the HARTS real-time distributed system. It features a synthetic workload generator and supports memory, CPU, and communication faults. The Orchestra fault injector was designed to test distributed protocols by being inserted between layers in the target protocol stack. It, too, supports communication faults. NFTAPE is a fault injector where fault injection and fault trigger mechanisms are plugged into its architecture based on the nature of the study being conducted.

Existing network fault injectors are useful for many studies, but typically do not have the ability to inject faults based on the global state of the system under study. Furthermore, they do not support the ability to compute many fine-grained dependability and performance metrics, such as the ones computed in this work. Loki [14, 15, 16] is a fault injector for distributed systems, developed at the University of Illinois, that specifically addresses these

two issues. Loki's most distinguishing feature, relative to other fault injectors, is that its fault-triggering mechanism can be based on the global state of the system under study. That allows one to examine experimental scenarios that it may not be possible or easy to produce and/or reproduce without such a mechanism. Loki also features a measures language for extracting a variety of dependability and performance measures from global timelines that Loki generates for each fault injection experiment. A good illustration of Loki's abilities is the evaluation of the unavailability induced by the Ensemble group membership protocol [17]. That work quantifies group membership blocking characteristics for Ensemble using both single failures and correlated failures.

Loki was chosen to conduct our evaluation of Coda for the following reasons. First, its global-state-triggered fault injection mechanism enables us to perform the correlated network partitions, and guarantees that they are correctly injected. In particular, it enables us to inject correlated network partitions at specific phases of Coda's resolution process, which would be extremely difficult without such a triggering mechanism. Second, Loki's global-state-tracking ability and measures support allow us to obtain a wide variety of measures from the vast amount of experimentally obtained data. It is important to examine several different metrics in order to gain insight about the behavior of Coda.

1.2 Thesis Contribution

This thesis makes several contributions, as described below.

- A front end was developed for the Loki fault injection tool to provide manageability of fault injection studies and usability of Loki's back end components. The front end is also a key component of Loki's runtime, which is used during fault injection experiments.

- A network fault injector was built to plug into Loki's runtime. Among other things, this fault injector supports the ability to perform network partitions.

- A case study of the Coda distributed file system was conducted using the Loki fault injector. The experimental evaluation provides insight about the impact that correlated network partitions have on Coda's resolution techniques.

- During the experimental evaluation of coda, it became necessary and highly beneficial to extend Loki with a lightweight fault injection proxy. Four ideas motivated the proxy approach. First, it reduces the intrusion caused by Loki on the system under

study. Second, it supports two features (explained later) that assist in the creation of state machines. Third, it provides a means to inject both internal and external faults. Finally, it exploits Loki's fault injection mechanism to support a means to manipulate the system under study through workloads and stimuli.

- The Coda case study demonstrates how global-state-triggered fault injection can be used to inject correlated faults into fine-grained states of the system. Furthermore, those injections show that the reaction of a system under study can depend on the states in which the injections occur.

- Loki's measures support is used to compute several fine-grained measurements that provide insight in addition to that of more traditional metrics such as coverage.

- This thesis provides the design of a fault injection study for a real system, i.e., Coda. That design includes state machines, fault triggers, and measure definitions, as required by Loki, and uses the proxy's experiment control features and fault injection support.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides an overview of Loki and explains how it can be used to evaluate a system. Chapter 3 presents a lightweight fault injection proxy approach to using Loki; it is the approach used in the Coda case study. Chapter 4 is an overview of the Coda case study. Chapter 5 presents measures that were computed for the study, and examines experimental results. Chapter 6 presents the conclusions of the thesis.

CHAPTER 2

OVERVIEW AND USE OF LOKI

This chapter begins with an overview of Loki. The overview includes terminology definitions, a brief description of Loki's fault injection process, a review of the Loki runtime, and a review of measure estimation. The overview is followed by an explanation about how to use Loki to evaluate a system.

2.1 Loki Overview

As previously mentioned, an important technique for experimental evaluation of distributed systems is fault injection. Global-state-triggering is beneficial to distributed system fault injection because distributed systems can fail in subtle ways that are dependent on the global state of the system. Triggering faults based on global state is difficult because by the time one node in a system has updated another about its state, the first node may have changed state.

Loki is a global-state-triggered fault injector for experimental evaluation of distributed systems. It is motivated by the need to experimentally evaluate the intricate behaviors and characteristics that are inherent to distributed protocols and systems. It strives to maintain goals of minimal intrusion, high precision, and high flexibility. Loki has been shaped by a few key ideas, including the concepts of optimistic synchronization, state abstraction, and separation of fault injection mechanisms from policy, as well as the observations that offline analysis is sufficient for fault injection and fault triggers depend on only a portion of the global state.

Optimistic synchronization is a technique by which faults are optimistically injected based on Loki's current local views of the global state at each node in the system under study. Loki assumes that each local view of the global state is always correctly aligned with the actual global state. This eliminates the need to synchronize the distributed system at every state

change in order to know the global state; such a synchronization would be extremely intrusive to the system under study. However, optimistic synchronization can lead to incorrect fault injections performed when the system is in the wrong global states. However, offline analysis can be used to negate this problem. Using an offline clock synchronization algorithm, Loki is able to create a global timeline from local timelines. The global timeline can then be used to filter out experiments containing incorrect fault injections. Offline analysis is sufficient for fault injection because fault injection is a process of postexperiment evaluation that never requires results during a fault injection experiment. Thus, optimistic synchronization is used to reduce intrusion, and offline analysis maintains high fault injection precision.

In Loki, the global state of the system is represented by a vector of the local states at each node in the distributed system. The local state abstraction is tracked by a state machine. State machines are specified at the granularity that is appropriate for fault injections and measurements, offering the user a great deal of flexibility in experiment design. During experiment execution, Loki takes advantage of the fact that in general, fault triggers depend on only a portion of the global state. Loki uses this observation to optimize the amount of state update traffic that it creates. Loki only tracks the part of the global state required at each node and sends the state updates only to nodes that require them, thereby reducing intrusion.

The last key concept is that Loki separates the mechanism used to carry out fault injection from the policy of fault injection. The mechanism used to carry out fault injection constitutes application-independent tasks such as running of experiments, global-state tracking, data collection, and offline analysis. On the other hand, the policy of fault injection pertains to application-dependent specifications such as state machine specification, fault trigger specification, and fault type (e.g., crash fault, network failure, or data corruption). By separating the two, Loki maintains a consistent mechanism for fault injection, so an experimenter is able to concentrate on the design of his/her fault injection campaign and is not restricted in his/her choice of fault types.

In addition to a fault injection methodology based on the above principles, Loki provides a flexible measurement language. The measurements are computed in a statistically correct manner from the global timelines constructed during the offline analysis. This is a significant aspect of Loki because it provides a practical manner in which to consolidate the results of an experiment. Without such a statistically based measurement language, the results of fault injection experiments would be extremely difficult to interpret. The usability of Loki is also enhanced by an extensive graphical user interface. It aids in the specification of fault injection campaigns, execution of campaigns, and obtaining of desired measures from campaign results. In addition to being a management console, the graphical interface also

provides important runtime functionality.

2.1.1 Terminology

- *State and state machine specification:* State is a fundamental concept to Loki because fault injections are performed based on a partial view of the global state of the system under study. In Loki, the user defines a state machine specification for each component of the distributed system. The execution of a component defines *local events* that trigger transitions between states in the component's state machine specification. These state machine specifications are used by Loki to track the *local state* of each of the components. The user is given the freedom to define each component's state machine at the level of abstraction that is appropriate for his/her experiments. The *global state* of the system is the vector of the local states of all of the components in the distributed system.

  It is important to note that Loki leaves the *granularity* of state machine specifications to the discretion of the user. It does so because state abstraction depends on the system under study as well as the desired type of evaluation. For instance, when evaluating recovery protocols, the user may want to compare different recovery techniques by measuring how much time each protocol takes to execute, or he/she may want to examine different steps of a single protocol in an attempt to improve its efficiency. In the first case, each recovery protocol may be modeled in a state machine as a single state. In the second case, each step or mini-step of the recovery protocol could be represented as a state.

- *Node:* Basic components of the system under study are combined with Loki runtime code to form a node. Each of these basic components has a corresponding *state machine specification* and *fault trigger specification*. Fault injections, recording of timeline data, and state tracking are performed at the level of the node.

- *Fault trigger:* Each fault that is to be injected during a fault injection experiment requires a fault trigger that indicates under what conditions the fault is to be injected. This trigger condition is based on a portion of the global state of the system under study. It is expressed as a Boolean expression on the local states of the node in the system. The literals of the expression are state-machine/state pairs. Literals can be negated ('`~`'), or combined using conjunctions ('`&`') and disjunctions ('`|`'). A fault is injected into a node if the node's view of the global state changes in a manner such that the trigger transitions from false to true. For example, `((sm_a:blue)|(sm_b:red))`

will trigger a fault to be injected if at some point in time state machine `sm_a` is not in the `blue` state and state machine `sm_b` is not in the `red` state, and then at some later point in time state machine `sm_a` transitions to the `blue` state and/or state machine `sm_b` transitions to the `red` state. As previously mentioned, the trigger is completely independent of the fault type to be injected. It simply determines when a fault is to be injected.

- *Partial view of global state:* A partial view of global state is the "interesting" part of the global state that is sufficient for performing fault injections at a node based on the node's fault triggers. It is a vector of local states from nodes on which fault triggers depend.

- *State change notification:* A state change notification is generated when a state machine changes state, and is sent to a list of state machines specified by the user. This list should be generated such that nodes in the system can maintain the necessary partial view of global state to be able to correctly perform the required fault injections. This list is determined by the user based on fault triggers and state machine specifications. This allows Loki to minimize its network traffic by sending only required notifications.

- *Fault injection campaign, study, and experiment:* In order to fault-inject a distributed system, the user designs a fault injection campaign. It consists of one or more studies. At the study level, the distributed system is broken into nodes. It is left to the user to divide the fault injection process into campaigns and studies. However, it is desirable for a study to consist of a set of similar fault injections and for a campaign to consist of a set of correlated studies. Loki provides a measure interface to interpret fault injection data at both the study and campaign levels. In order to obtain statistically significant results, each study and its corresponding fault injections are performed several times. Each run is called an experiment.

2.1.2 Brief description of Loki's fault injection process

Two processes are necessary to conduct fault injections in Loki. The first process is the specification process, during which the user organizes fault injection campaigns and studies, and decides on the number of experiments for each study. For each study, he/she partitions the distributed system into nodes and specifies state machines and fault triggers. Using Loki's measure specification language, the user also specifies measures to be computed from the results of the experiments. The Loki front end, described in detail in Section 2.2, provides a convenient interface through which the user can provide Loki with these specifications.

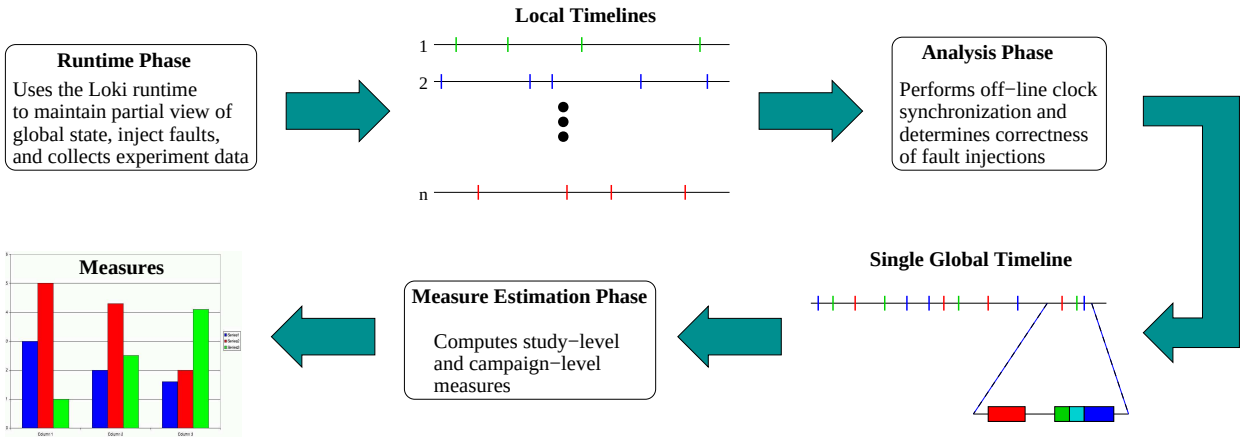


Figure 2.1 Using Loki to Evaluate a System

After the specification process, the evaluation process is performed. As shown in Figure 2.1, the evaluation process is divided into three main phases, namely the *runtime phase*, the *analysis phase*, and the *measure estimation phase*. Each of these phases is completed, for each experiment in every study in a campaign, before the next phase is begun. At the end of this evaluation process, the user has obtained the desired performance and dependability measures.

During the runtime phase, the experiments are performed. Each run involves execution of the distributed application, tracking of the partial view of global state for each of the nodes, checking of whether any of the fault triggers are satisfied (and, if they are, injection of the corresponding fault in the appropriate node), and collection of the required experiment data. These tasks are performed by the Loki runtime. Also, synchronization timestamps are collected for each study for use in the analysis phase.

During the analysis phase, the Loki analyzer performs offline clock synchronization using the synchronization timestamps collected during the runtime phase. It converts the local times of recorded experiment data into a single global timeline. The analyzer also checks the fault injections for correctness using the global timeline. The details of offline analysis and formulation of global timelines are not reviewed here. They can be found in [14, 18]. Following the analysis phase, the Loki measure estimator uses the global timelines from the experiments with correct fault injections to estimate user-specified measures in the measure estimation phase. From the measure estimation phase, experiments containing incorrect fault injections can be discarded as required. The fault injection process ends with the measure estimation phase, after which the user has obtained the required measures from the fault injections.

2.1.3 Review of the Loki runtime

As previously mentioned, the Loki runtime is used during the runtime phase of the fault injection evaluation process to conduct fault injection experiments. It is responsible for the following tasks:

- Coordinating the start and end of experiments,
- Monitoring each application node for local events,
- Maintaining a partial view of the global state at each node (i.e., it uses the local events to track local state and sends state change notifications among nodes to track the required global state),
- Checking the fault triggers at each change of the partial view of global state,
- Injecting the corresponding fault if a fault trigger transitions from a false to a true on a state change (in the partial view),
- Monitoring each node for crashes, and
- Recording data regarding the interesting events in the system, particularly the local occurrence times of the events. These "interesting" events include the start and end of fault injection experiments, local state changes caused by local events, fault injections, and any application node crashes and restarts.

The Loki runtime is composed of three major pieces. One piece is attached to each node in the system under study and is referred to as the node's *runtime*. The other two pieces are *local daemons*, one per host, and a *central daemon*. The architecture for the runtime is shown in Figure 2.2. The figure illustrates three hosts with corresponding local daemons, four nodes with corresponding runtimes, and the central daemon.

The original Loki architecture comprised only the node runtimes. Each of these runtimes was made of five components, namely the *state machine*, *state machine transport*, *parser*, *probe*, and *recorder*. All of the components except the probe are independent of the system under study, and the probe is implemented by the user. The *heartbeat provider*, another system-independent component, was added later when the local daemons and central daemon joined the architecture, and will be covered following a discussion of the components of the original architecture.

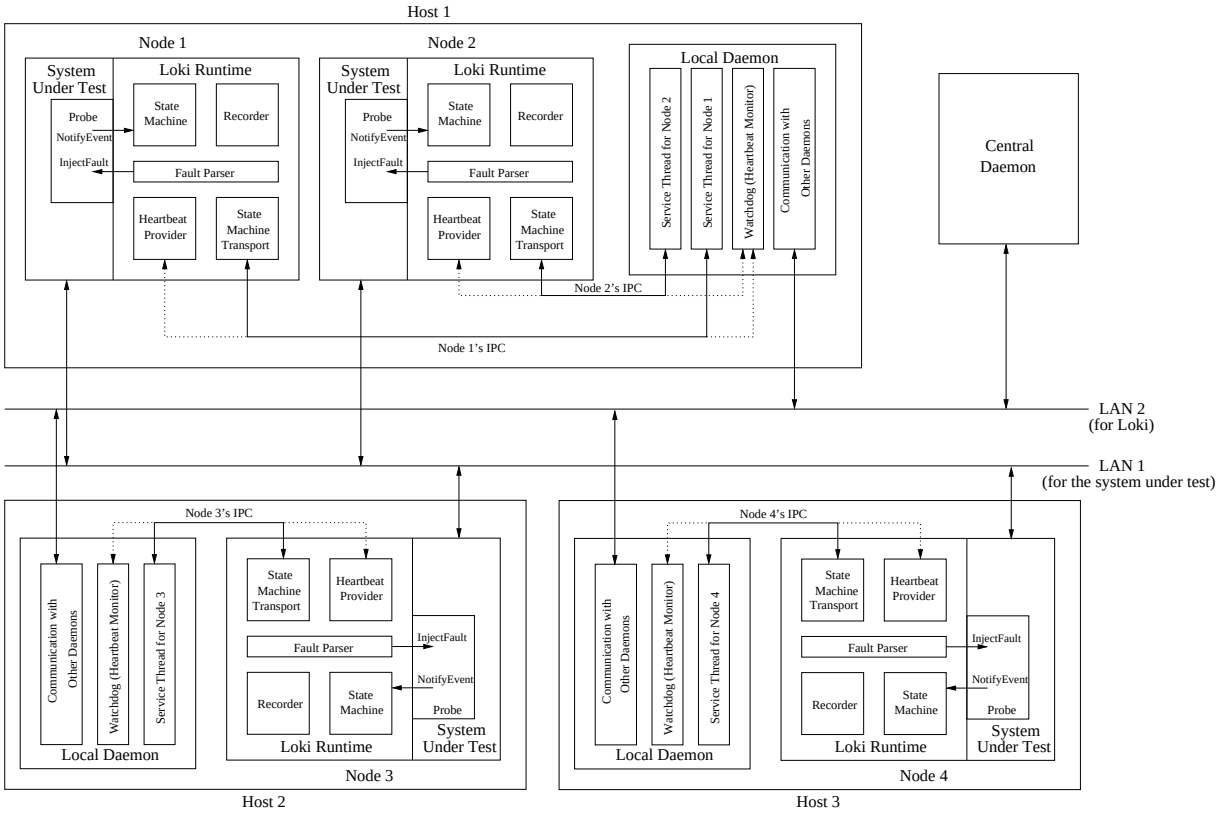


Figure 2.2 Loki Runtime

The purpose of the state machine[1] is to track (1) the local state of a node, and (2) the partial view of global state that is necessary for fault injections into the node. When local events occur, the state machine is notified by the probe. It uses these event notifications along with the state machine specification to track the local state of the node. The new state is a function of the old state and the local event notification. Also, the state machine maintains a partial view of global state by tracking the state of remote nodes as necessary for local fault injections.

The state machine makes use of its state machine transport to send and receive notification messages. The purpose of the state machine transport is to provide a simple interface for sending and receiving notifications between nodes. It performs this task by hiding the intricacies of the communication system from the other runtime components.

Whenever the partial view of global state changes, the state machine informs the fault parser. The fault parser uses this information to determine if any faults should be injected. A fault is injected when a change in the partial view of global state causes its Boolean trigger

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

to transition from false to true. If a fault is to be injected, the fault parser instructs the probe to inject the fault.

As already mentioned, the probe is responsible for notifying the state machine when local events occur and performing fault injections when instructed to do so by the fault parser. The probe is different from the other components in that it is system-dependent. The user implements it when instrumenting a distributed system for use with Loki. Details concerning the implementation of the probe and instrumentation can be found in Section 2.2.

The final component of a node's runtime that was included in the original Loki architecture is the recorder. It records information relevant to a fault injection experiment. This information includes occurrences and occurrence times for local events, state changes, and local fault injections. At the conclusion of an experiment, this data forms a local timeline. Whenever possible, Loki uses hardware clocks to improve precision and lower intrusion.

The original Loki architecture was later modified because it could not handle dynamic node entry into and exit out of the system under study [16]. In other words, all nodes in the system had to be started at the beginning of an experiment, and during the course of the experiment, nodes could not enter or leave the system. In order to support dynamic nodes, the Loki architecture was altered through the addition of daemons that are independent of nodes, to detect the entry and exit of nodes during the execution of an experiment. However, the complications of dynamic nodes are not limited just to detection. Another problem is that dynamic nodes disrupt the communication links between the runtimes at each node. Establishing new communication links and removing old links can be intrusive to the system under study. Thus, the daemons also serve as a level of indirection for communication between nodes.

Two types of daemons are used by Loki, *local daemons* and a single *central daemon*. A local daemon is run on each host on which nodes from the system under study may reside. A local daemon is responsible for coordinating experiment execution on its host, routing notification messages for state machine transports, monitoring local nodes for crashes, and locally managing dynamic node entry and exit. The central daemon coordinates the local daemons to perform runtime tasks. These tasks include starting up components required for an experiment, aborting an experiment in case of abnormal circumstances (e.g., local daemon crashes or experiment hangs), and determining when an experiment is complete. In order to perform the above tasks, each local daemon and the central daemon maintain a reliable, ordered communication link. The local daemons communicate to local nodes through IPC.

In addition to the daemons, a heartbeat provider was also added to each of the node runtimes. The heartbeat provider sends heartbeat messages to the local daemon on its host, to let the daemon know that its runtime is alive and properly running.

In the current implementation of the Loki runtime, the communication links between the central daemon and local daemons are TCP links, while the communication between a local daemon and local nodes is performed using shared memory and semaphores. Also, in the current implementation, nodes, local daemons, and the central daemon all run in their own unique processes. The Loki runtime adds three threads to each node. One thread hosts the state machine transport, another hosts the parser, and the final thread hosts the heartbeat provider. The state machine and the probe execute within the application's thread(s). Local daemons are made up of four main threads, plus additional threads servicing communication channels. One thread accepts connections from new and restarted nodes, one communicates with other local daemons, one communicates with the central daemon, and the final thread monitors heartbeats from local nodes. Node runtimes and local daemons are implemented using the C programming language. Details concerning the implementation of the central daemon can be found in Section 2.2. Below is a description of the Loki runtime operation.

- *Start of an experiment:* At the start of an experiment, the central daemon launches each local daemon. During initialization, the local daemons connect to each other and to the central daemon using TCP. The central daemon then instructs the local daemons to start nodes that the user has specified should be started at the beginning of the experiment.

- *Node startup:* Because of the support that the daemons and heartbeat provider afford, a node can be started either by the central daemon at the start of the experiment, or by the application at any point during execution of the experiment. Either way, when a node starts, its state machine transport sends a connection request, on a well-known IPC channel to its local daemon. Upon receiving the connection request the local daemon creates a new IPC channel to communicate with the node, and spawns a new thread to service the new communication channel. Also, the local daemon notifies all other local daemons that a node has entered the system. This also provides local daemons with routing information to the node entering the system.

- *Normal operation:* When node x changes state and other nodes need to be notified, x's state machine transport sends a state change notification to its local daemon along with a list of the nodes to be notified. Upon receiving this request, the local daemon looks up the routing information for each node that is to be notified. If a node is remote, then x's local daemon forwards the notification over TCP to the receiving node's local daemon. Even if more than one of the nodes to be notified are located on the same host, only one notification message need be forwarded between the two local daemons.

After the notification has been forwarded by x's local daemon to other local daemons, the receiving local daemons use IPC to forward it to the appropriate receiving nodes. If a receiving node is located on the same host as x, then x's local daemon simply cuts out the TCP step and forwards the notification using IPC. If a notification is to be sent to a nonexistent node, the notification is discarded and a warning message is written to a log file. Locally, each node's parser checks all of its fault triggers and injects the required faults whenever its partial view of global state changes. Each node's recorder records the times and occurrences of local events, state changes, and fault injections to a local timeline.

- *Normal node departure:* When a node exits normally, it notifies its local daemon, which in turn notifies all other local daemons so that each daemon will have an updated view of which nodes are running and which are not.

- *Node crash:* Loki employs two methods to detect node crashes. First, if a node's signal handler[2] is invoked before it crashes, the handler will delete the IPC channel between itself and its local daemon. The local daemon can detect this event. The second method involves a timeout on heartbeat messages sent from a node to its local daemon. The local daemon serves as a *watchdog* monitoring heartbeat messages. The value of the timeout is adjustable when the node's state machine is specified by the user. After a crash is detected, the local daemon writes a crash event to the crashed node's local timeline and notifies all local daemons about the crash

- *Node restart:* First, the node determines whether it has been restarted. It does so by having its state machine check its local timeline. Since the node could start on a new host, the local timeline is stored on a network file system. If the node is restarted, it writes a restart event to its local timeline. The node then follows the normal procedure for starting up. In addition, the restarted node reconstructs its partial view of the global state by obtaining state updates from all other nodes.

- *End of an experiment:* There are two ways for an experiment to end. The first way is for the application to exit on its own. The second way is through an application timeout that is defined by the user. If the timeout value is reached, the central daemon instructs each local daemon to kill all of the nodes located on its host. When a local daemon detects that all of the nodes have exited or crashed, it sends an "experiment complete" message to the central daemon. When the central daemon receives "exper-

[2]This is why the user must pay special attention when overriding the signal handler in an application.

iment complete" messages from all local daemons, then the central daemon declares that the experiment is over and the next experiment can begin.

2.1.4 Review of measures

Another important aspect of Loki is its ability to estimate user-defined performance and dependability measures. Loki uses statistically accurate techniques to compute estimators for data collected during the experiment process. The measure specification language and the estimators that Loki computes will be reviewed. They were originally presented in [16] along with details for the computation of estimators. An abbreviated syntax is used in this thesis to improve readability.

Measures in Loki are computed at the study level and the campaign level. Study-level measures are computed from the data collected during experiments from a particular study. Campaign-level measures are computed by combining the study-level measures from studies in a particular campaign. The breakup of measures into these two levels is natural because a study consists of experiments involving similar fault injections, and a campaign consists of correlated studies. For example, a user may want to compute an overall fault coverage of a system. One way to do so is to design a campaign in which each study involves the injections of one injection scenario. A study-level coverage measure can be computed for each study with respect to its injection scenario. The study-level coverages can be combined using their occurrence frequencies, perhaps obtained from empirical data; thus, using a campaign-level measure, an overall system coverage can be determined.

2.1.4.1 Study-level measures

A study-level measure is computed from the global timelines of a subset of the experiments executed for a study. In order to arrive at that subset, the measure applies a series of filters to the global timelines to narrow down the experiments to a group that has certain properties. A measure computation function is then applied to this remaining subset of experiments to obtain the study-level measure. The three ingredients involved in this procedure are predicates, observation functions, and subset selection. They are applied as an ordered sequence of triples, i.e., ((subset selection$_1$, predicate$_1$, observation function$_1$), ... (subset selection$_n$, predicate$_n$, observation function$_n$)), where $n \geq 1$. The three components of the triples are defined below, and then an example study-level measure is given.

Predicates are Boolean expressions used to test a global timeline to determine when it meets certain conditions. At a particular point in time a predicate will evaluate to a true or false value. This is known as a *predicate value*. When the predicate is applied over a whole

timeline, it produces a function of true and false versus time, known as a *predicate value timeline*. Predicates are made up of conjunctions ('&&'), disjunctions ('||'), and negations ('!') of queries over the states and events of the state machines involved in the experiment.

There are seven types of state machine queries: (state machine, state), (state machine, state, time), (state machine, state, start time, end time), (state machine, event), (state machine, event, start time, end time), (state machine, state, event), and (state machine, state, event, start time, end time). The first three queries are true if the specified state machine is in the designated state. The first query is applied to the entire global timeline. The second query is applied at a time instant. The third query is applied over a time interval. The fourth and fifth queries are true whenever the designated event occurs in the specified state machine over the entire global timeline or an indicated time interval, respectively. The final two queries produce a true result whenever the specified state machine is in the specified state and the given event is generated at that state machine over the course of the entire experiment or a time interval, respectively.

In addition to those queries, some higher-order predicate logic is supported. For details see [19]. Although this list is not necessarily comprehensive, the higher-order logic includes forall(list, [x, predicate]), forany(list, [x, predicate]), foratmost(list, num, [x, predicate]), and foratleast(list, num, [x, predicate]). Each of these quantifiers binds a variable x for use in the specified predicate. The first parameter to each quantifier is a list of objects that form the domain of x. forall is a universal quantifier, and forany, foratmost, and foratleast are existential quantifiers.

An *observation function* is used to map a predicate value timeline to a single value called an *observation function value*. There are several predefined observation functions: count(transition direction, transition type, start time, end time), outcome(time), duration(predicate value, n, start time, end time), instant(transition direction, transition type, n, start time, end time), and total_duration(predicate value, start time, end time). In these functions, transition directions are either UP, $DOWN$, or $BOTH$. Transition types are either $IMPULSE$, $STEP$, or $BOTH$. Predicate values are $TRUE$ or $FALSE$. In addition to the predefined functions, a user can define his/her own observation functions by combining predefined observation functions with C syntax that can be compiled by a standard C compiler.

The five predefined observation functions allow a user to extract information from a predicate value timeline. count($\cdot$) returns the number of times a transition in the specified direction and of the specified type occurs during a particular time interval. outcome($\cdot$) returns 1 if the value of the predicate value timeline is true at the specified time instant; otherwise it returns 0. duration($\cdot$) returns the amount of time (in ms) that the predicate

value timeline has the specified predicate value after the n^{th} transition from false to true, during the given time interval. instant($\cdot$) returns the time instant from the predicate value timeline, where the n^{th} transition of the specified direction and type occurs during the given time interval. total_duration($\cdot$) returns the length of time for which the predicate value timeline has the specified predicate value during a given time interval.

Subset selection is a Boolean test performed on the observation function values for an experiment to determine if the study measure should continue to consider the experiment. The test is created from standard mathematics and previous observation function values for an experiment. If the test is true for an experiment, then the experiment continues to be included for the measure; otherwise, it is discarded. It should be noted that the first subset selection is predefined to include all experiments.

Each (subset selection, predicate, observation function) triple is named so that its result can be used by subsequent triples. The observation function value for an experiment at the end of a previous triple can be used in the application of a new triple, through reference to the name of the previous triple in the new triple's definition. There is also a function that can be used in triples to check the injection status of a fault at a given state machine [19]. This function is faultcheck(state machine, fault, injection status). The injection status can be $CORRECT$, $INCORRECT$, $NO\_INJECTION$, or $INJECTION$.

Study-level measures are computed by applying each triple in succession as demonstrated in Figure 2.3. The first subset selection includes all of the experiments. Therefore, the first predicate is applied to each experiment, creating a predicate value timeline for each experiment. The first observation function is then applied to each of the predicate value timelines that were produced by the first predicate. The observation function assigns an observation function value to each experiment. Next, the second subset selection is applied, and experiments that do not meet its criteria are discarded. The second predicate and observation function are applied to the remaining experiments. This produces new observation function values for all experiments that have not been discarded. This process continues until the final observation function is applied. It results in *final observation function values* for experiments that have passed all of the subset selections. The final observation function values make up the sample for the study-level measure.

Figure 2.4 demonstrates the application of a predicate and an observation function to an experiment. First, the predicate is applied to the experiment's global timeline. After being applied, the predicate looks for periods of time when state machine `sm_a` is not in the `blue` state and occurrences of the event `event_o` in state machine `sm_b`. The result is a predicate value timeline that is true between times 15.0 and 20.0 and that has an impulse at time 22.0. Then the measure applies an observation function, which counts the number of times

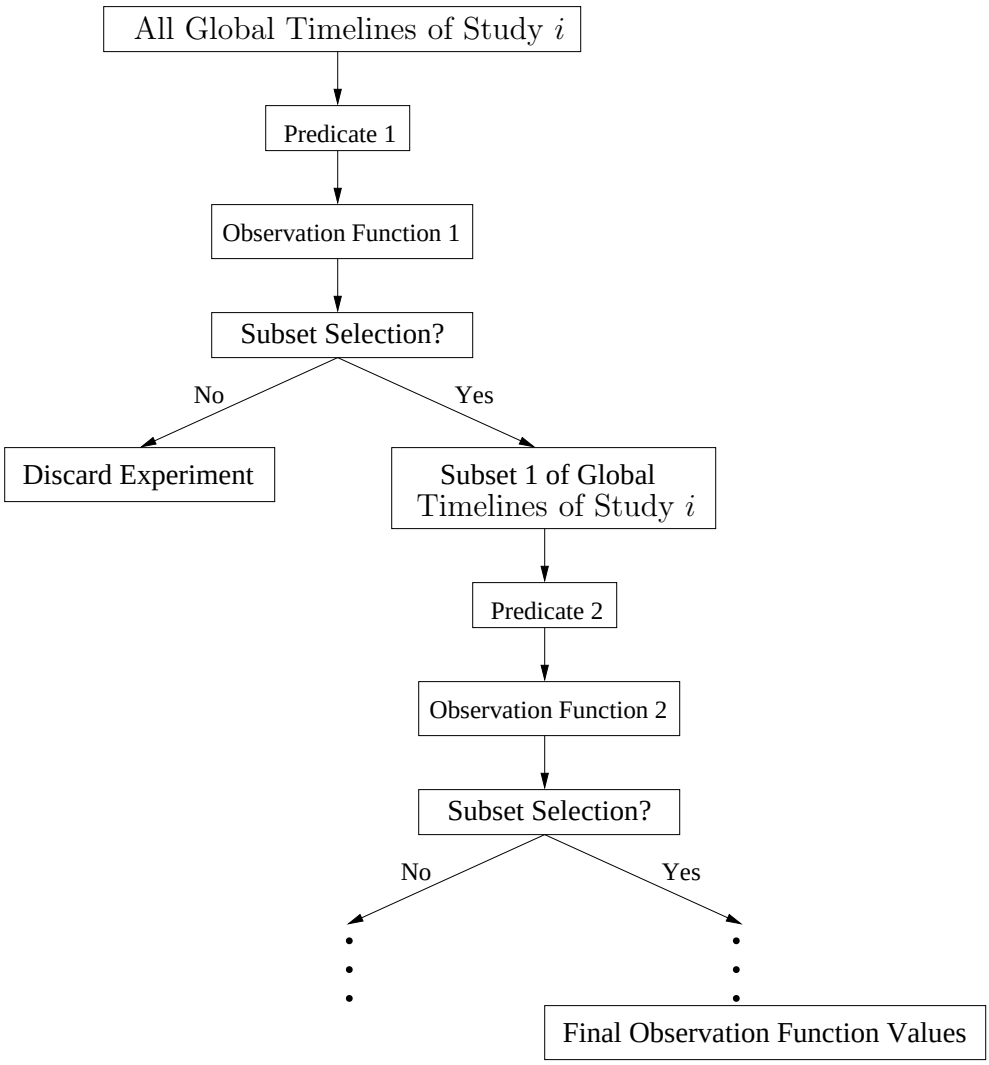


Figure 2.3 Study-Level Measure Procedure

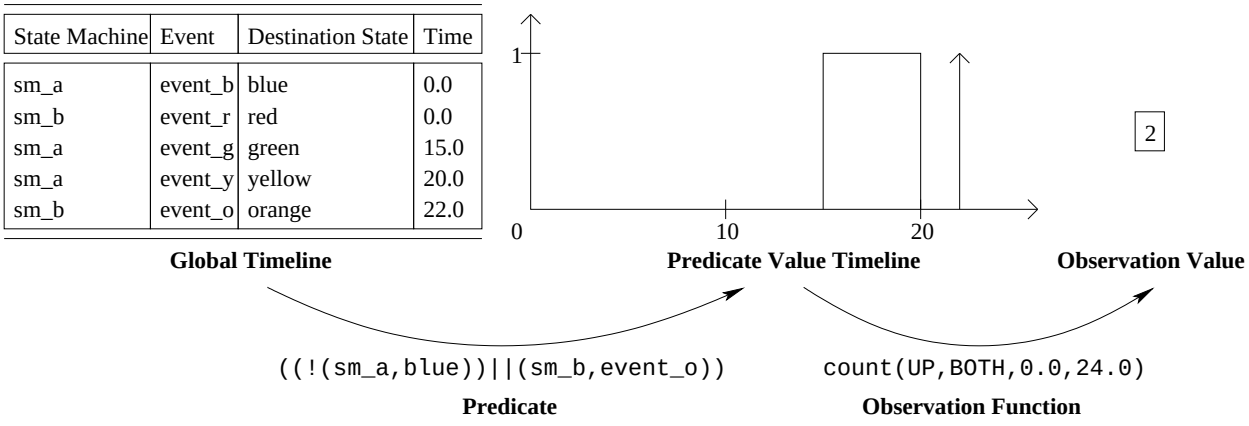


Figure 2.4 Study-Level Measure Example

an up transition of either a step or impulse type occurs. From the predicate value timeline it is apparent that the observation function value is 2. This value could be returned as the final observation value for the experiment, or a subset selection function could be applied. For example, a user might filter out all experiments whose observation value is less than 5, in which case the experiment in this example would be discarded.

2.1.4.2 Campaign-level measures

Campaign-level measures are defined by combining study-level measures. Loki supports three types of campaign-level measures: simple sampling, stratified weighted, and stratified user. In theory, a campaign-level measure is completely characterized by its distribution. However, it is not possible to determine the distribution in general. Fortunately, knowledge of the moments is equivalent to knowledge of the distribution function because it is theoretically possible to exhibit all the properties of the distribution in terms of the moments [20, p. 108-109]. In practice, the properties obtained from the first four moments closely approximate the properties of the real distribution. All of the campaign-level measure types are described below. However, details concerning how the statistical properties of the measures are computed are found in [16].

Simple sampling measures are used when a measure does not require that the final observation function values from different study-level measures be differentiated. The user selects a group of study-level measures to be included in the simple sampling measure, and the observation function values from each study-level measure are placed in a single sample, i.e., they are all instances of the same random variable. Several statistical properties are computed for simple sampling measures, including the first four moments, skewness coefficient, kurtosis coefficient, and several percentiles for various α-values.

Stratified weighted measures are used when a user wishes to linearly combine several study-level measures. The user specifies the linear combination with a weighting function. This type of measure considers the results of each study-level measure to be an independent sample. Moments are computed for each sample, and their estimations are linearly combined using weights derived from the weighting functions. Calculation of the moments is linear for a linear combination of random variables and is characterized by the property $f(ax + by) = cf(x) + df(y)$, where x and y are random variables, $f(\cdot)$ is a function for calculating a moment, a and b are weights, and c and d are weights derived from a and b. Here it is assumed that the powers of random variables representing the final observation function values are independent across studies. Again, the first four moments, skewness coefficient, kurtosis coefficient, and percentile points for various α-values are computed for stratified

weighted measures.

The final type of campaign-level measure is the stratified user measure. *Stratified user measures* allow the user to combine study-level measures with a user-defined function other than a linear combination. Unfortunately, the statistical properties of such measures are not well-defined. Calculation of the first four moments is a nontrivial task for an arbitrary function. Therefore, the only result Loki produces for this type of measure is the actual campaign measure, which is computed by substituting the means of the final observation function values for study-level measures used in the user-defined function. Unfortunately, the campaign measure may have no statistical meaning.

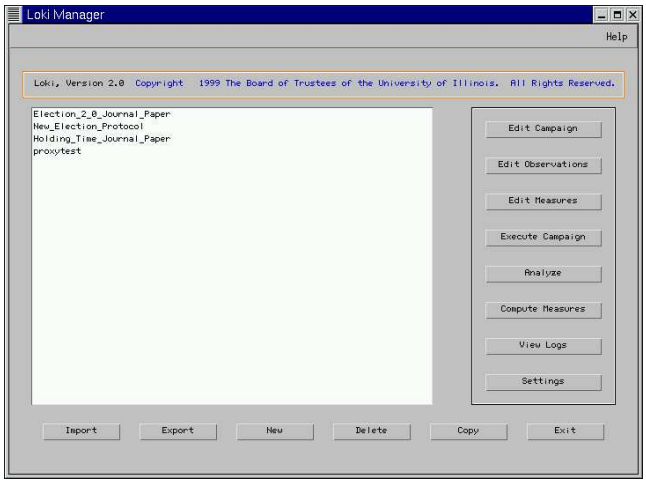
2.2 System Evaluation Using Loki

The Loki front end supplies a graphical interface for the fault injection tool. It provides a means to specify, execute, and process results for fault injection campaigns. In addition, during the execution of campaigns, it provides the central daemon functionality described in Section 2.1.3. The main goals in developing the front end were manageability of fault injection campaigns and usability of the runtime, analysis, and measure back ends. The Loki Manager, as seen in Figure 2.5(a), controls the main functionality of Loki. The task panel on the right side takes a user through the steps of a fault injection campaign, including campaign specification, measures specification, campaign execution, and offline processing. Please note that previous and future publications of the material in this chapter may vary, as the implementation and design of Loki change.

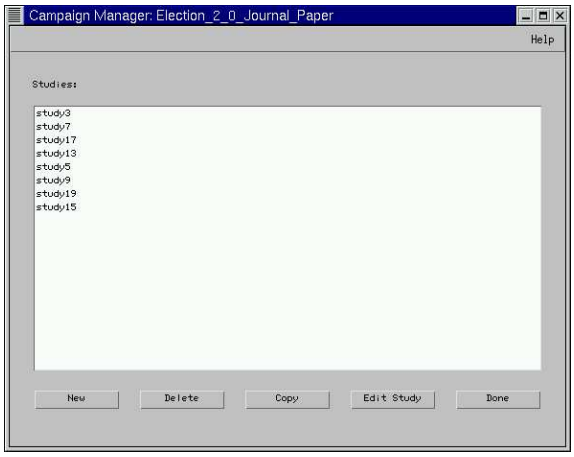
2.2.1 Campaign specification

The first step in campaign specification is to create and edit a campaign using the Loki Manager (Figure 2.5(a)). When editing a campaign, a Campaign Manager similar to the Loki Manager pops up and allows a user to create the campaign's studies. As mentioned in Section 2.1.1, a campaign is composed of correlated studies and a study consists of a set of similar fault injections. After each study is created, it needs to be defined. This is done from the "Edit Study" button on the Campaign Manager (Figure 2.5(b)), which launches the Study Manager (Figure 2.5(c)).

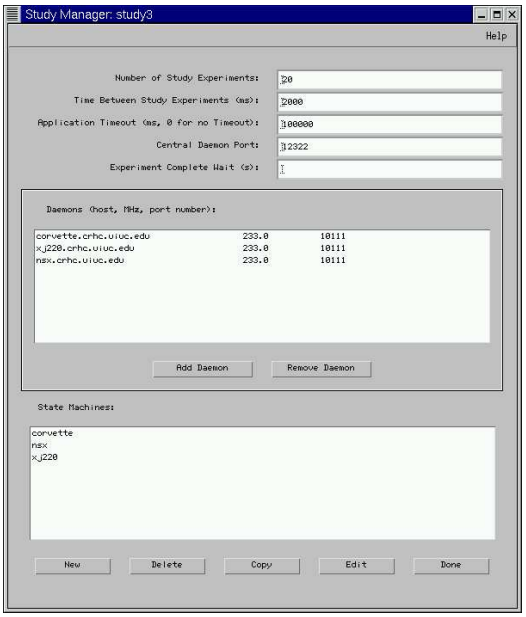
There are three elements in the Study Manager. The first comprises general study parameter values, including the number of experiments to be run, the time to wait between experiments, an application timeout value for each experiment, a port by which local daemons can contact the front end (which acts as the central coordinating daemon), and an



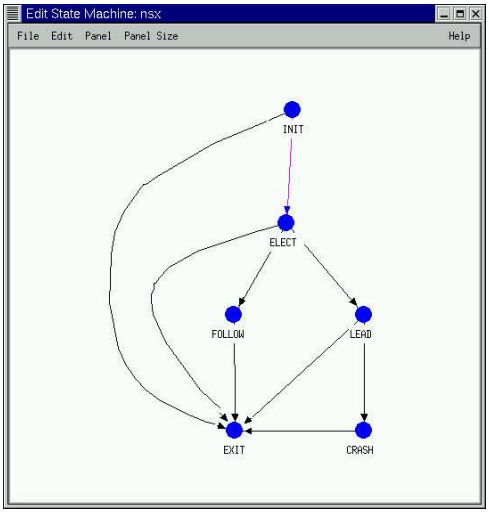
(a) Loki Manager



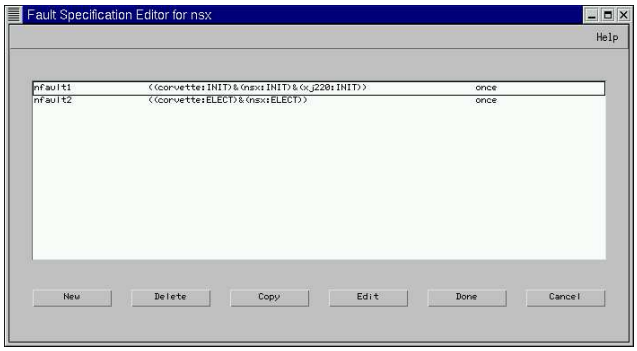
(b) Campaign Manger



(c) Study Manager



(d) State Machine Editor



(e) Fault Specification Editor

Figure 2.5 Loki Campaign Specification Screen Shots

experiment completion wait value. The application timeout value is used to limit the length of an experiment because many experiments may not have an explicit end. The other option is to define a timeout value of 0 (indicating no timeout value), and have the system under study use its own mechanism to determine when to terminate. The experiment completion wait value is the amount of time that local daemons give a node to shut itself down after being asked to do so by the daemon.

The second element in the study manager is the daemon list. A daemon must be registered for each host containing an instrumented process in the distributed system under study. The "Add Daemon" button allows the user to register a local daemon. Registration requires a host name, the clock speed of the host's processor, and a port number for communication. The clock speed is required so that timelines can be converted from clock ticks to μs. The last element of the Study Manager is the state machine list. Each node in the system from which state will be tracked or into which a fault will be injected during an execution of the study, regardless of whether it shares an executable with another node, must be represented in the state machine list. After a state machine is added to the list, it must be defined via the "Edit" button. The definition of a state machine includes both a state machine specification and a fault specification. The "Edit" button launches an editor for both.

The State Machine Editor is displayed in Figure 2.5(d). It is responsible for generating a state machine specification from a local state abstraction entered by the user. The state machine specification is employed by the Loki runtime when it is tracking the partial view of global state required by the study. From the State Machine Editor menu bar, the user can start a properties dialog to specify four parameters for the state machine, including a host name, an application name, application arguments, and a heartbeat timeout value. The host name indicates the host on which the node associated with the state machine should be started when an experiment is run. The absence of a host name indicates that the node should not be started by Loki at the beginning of an experiment. The application name identifies the instrumented executable that, when executed, is a node of the system under study. The application arguments are those that should be passed to the application when it is started. The application heartbeat is used by the local daemons to determine whether the node is alive. If a node does not send a heartbeat within the specified timeout, then it is assumed dead.

The majority of the State Machine Editor is composed of a state machine canvas on which the local state abstraction is defined. Three components are used on the state machine canvas, including states, event-triggered state transitions, and comment boxes. States are represented on the canvas by named vertices. In addition to naming a state, the user selects a list of state machines to be notified when the state machine enters the state. This list

helps Loki maintain the partial view of the global state. If one state can transition to another, a directed edge joins the two states. The edge represents an event-triggered state transition. When creating a state transition, the user provides the name of the triggering event. Comment boxes are also used on the state machine canvas to document additional information about the state machine.

The counterpart of the State Machine Editor is the Fault Specification Editor, seen in Figure 2.5(e). It creates a fault specification containing the names and fault triggers for faults that are to be injected into the node for which the editor was launched. The names given here are the same as those used for instrumentation of the system under study. The fault triggers are defined with two parameters. The first parameter is a Boolean fault expression pertaining to some partial view of the system's global state, where the partial view of the global state is the one into which the corresponding fault should be injected, as described in the Fault Trigger definition in Section 2.1.1. The second parameter of the fault trigger is the indication of whether the fault should be injected "once" or "always." If the "once" option is set, then the fault will be injected the first time the expression transitions from false to true. If the "always" option is set, then the fault will be injected every time the expression transitions from false to true.

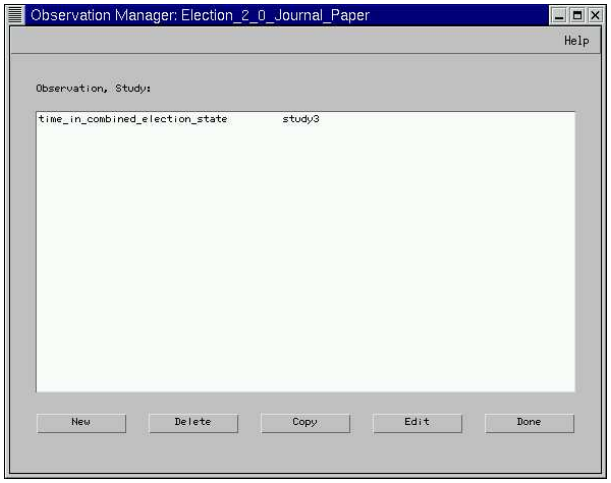
2.2.2 Measures specification

From the Loki Manager, the user defines study-level measures (which are referred to as *observations*[3] in the Loki front end) and campaign-level measures. When the "Edit Observations" selection is made from the Loki Manager, the Observation Manager in Figure 2.6(a) is launched. From that manager, observations are created and assigned to particular studies. When observations are defined, the Observation Editor (seen in Figure 2.6(b)) is started. It allows the user to name and define (subset selection, predicate, observation function) triples as described in Section 2.1.4. The "Commit" button is used to commit changes for a triple, while the "Compile" button saves the full observation to disk and compiles it into an executable. If there are compiling errors, then a dialog launches that reveals the compiling errors and pinpoints the parts of the observation specification containing those errors.

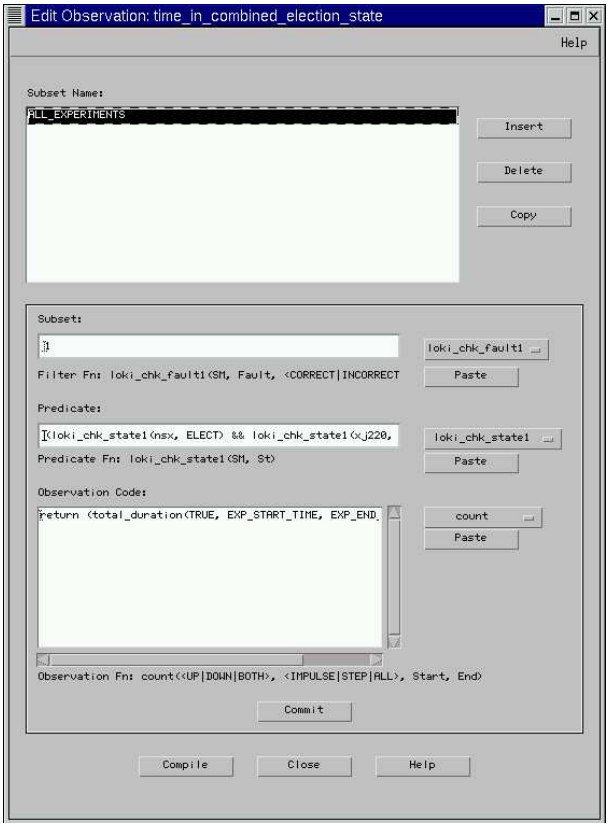
After the user has defined observations, he/she is ready to define campaign-level measures. From the Loki Manager, a Measure Manager[4] similar to the Observation Manager is invoked. The Measure Manager is displayed in Figure 2.6(c). Within the Measure Manager,

[3]In the Loki front end, the term *observation* is used for study-level measures (described in Section 2.1.4) to distinguish them from campaign-level measures (see Section 2.1.4).

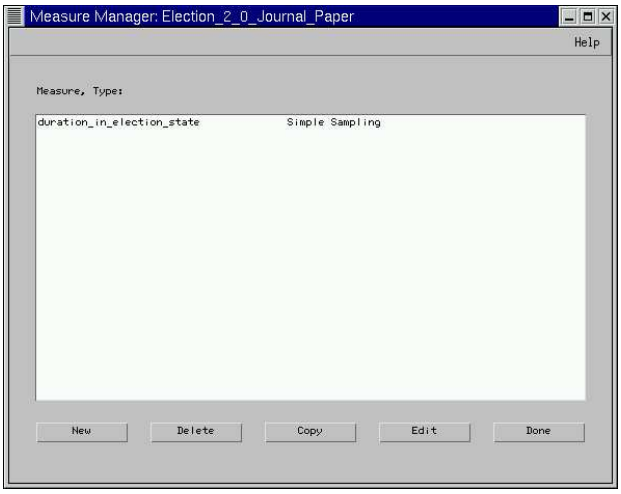
[4]In the Measure Manager and the Measure Editor, the type of "measure" that is examined is the campaign-level measure, described in Section 2.1.4.



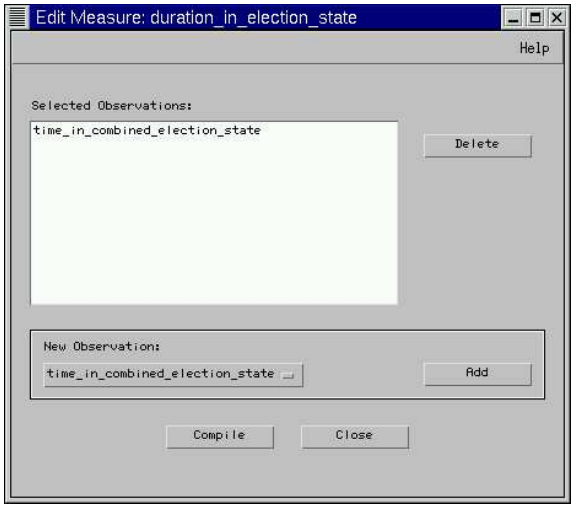
(a) Observation Manager



(b) Observation Editor



(c) Measure Manager



(d) Measure Editor

Figure 2.6 Loki Measures Specification Screen Shots

campaign-level measures are created and assigned one of three types (simple sampling, stratified weighted, or stratified user), as explained in Section 2.1.4. After a measure is created, it must be defined. If the measure is a simple sampling measure, then the observations that compose the measure are selected in the Measure Editor, seen in Figure 2.6(d). If the measure is a stratified weighted measure, then observations and weights must be selected. Finally, if the measure is a stratified user measure, then the Measure Editor can be used to compose a user-defined function to combine observation means. Again, the "Compile" button is used in order to compile the measure specifications into executables. As with the Observation Editor, if there are compiling errors then a dialog launches to help the user resolve them.

2.2.3 Loki range editor

Many times it is desirable to design several similar studies within a campaign. For instance, a user may want to perform studies that are identical except that the state into which a particular fault is to be injected is varied. However, maintaining each of the studies can be unnecessarily tedious, especially if they must all be modified in an identical manner. The Loki Range Editor and range studies have been designed to accommodate this and similar situations. The *Loki Range Editor* generates normal studies from a range study. A *range study* is designed like a normal study, but the user can use *range variables* in the specification of the study to be later replaced with a value from a range of values. The names of the range variables are preceded and followed by three underscores. In other words, any user input to a campaign specification in the format ___<name>___ is regarded as a range variable. Range studies are identified with a ".range" extension applied to the end of the study name. These studies are not executable. They merely make it convenient for a user to generate several similar studies.

After designing a range study, the user writes a *range specification* to tell the Loki Range Editor the values to substitute for range variables in the studies that it generates. Range specification files contain named sets of ranges for the range variables in a study. Each set should have definitions for all of the range variables in the study. The format for the file is as follows:

```
<setname-1>{
___var-1___ <variable definition>
...
___var-m___ <variable definition>
}
```

```
...
<setname-n>{
___var-1___ <variable definition>
...
___var-m___ <variable definition>
}
```

There are two types of variable definitions: a list and a numeric range. Lists are comma-separated and surrounded by parentheses, i.e., `(value-1,...,value-n)`. Numeric ranges are specified as triples in brackets. The parameters include an initial value, a maximum value, and an increment. Numeric lists have the form `[initial,max,+increment]`. In the future, support for other mathematical manipulations (in addition to increment) can be easily added. For each set, the Loki Range Editor generates a study for each combination of values contained within the set. The generated studies are named <studysubname>-<setname>-<*value*(__var-1__)>- ... -<*value*(__var-n__)> where <studysubname> is the range study name without the ".range" extension, <setname> is the name of the current set, and <*value*(__var-x__)> is the value for __var-x__ that has been substituted into the generated study. However, if the name of a range variable is prepended by 'I,' then its value is not used in the name of the generated study.

The Loki Range Editor is written in Perl and is invoked using `lokiRangeEditor.pl [OPTIONS] <Study> <RangeSpecification>`. The Loki Range Editor has also been extended to work on observations in much the same way it works on studies. The user enables it by specifying an "`-m`" option to the Loki Range Editor.

2.2.4 System instrumentation

Once the campaign has been fully specified, the system under study is ready to be instrumented. Four major steps, described below, are involved in instrumenting a system for fault injection. In the following descriptions, remember that a node is the basic component of the system under study into which faults may be injected and whose state is tracked.

1. *Implementation of faults:* The first step is to implement the faults to be injected. For nodes that require the injection of faults, the `injectFault()` method of the node's runtime probe must be coded with an implementation of those faults. The only input parameter to the `injectFault()` method is a fault name corresponding to the names specified in the Fault Specification Editor. The method returns the time at which the fault is injected. This technique of having the user implement the desired faults

provides a high degree of flexibility by separating the user's policy from the overall mechanism of fault injection.

2. *Event notification:* The second step is to indicate the occurrence of events that pertain to transitions in the state machine of a node. The user does so by inserting the probe's `notifyEvent()` method at appropriate places throughout the system under study to indicate when events occur. The parameters of `notifyEvent()` are the name of the event and the time at which it occurred. Event notification is required of nodes whose state must be tracked.

3. *appMain:* The next step is to use an `appMain()` method in place of the standard `main()` method to initiate an instrumented process. This is required for all instrumented nodes.

4. *Node invocation:* The final step is to modify the invocation commands for nodes. This change involves the arguments passed to nodes. In order for the Loki runtime to identify study parameters, locate the appropriate local daemon, and compute the location to write data to, the first three arguments to a node must be its corresponding *StudyFile*, the study's *DaemonContactFile*, and the current experiment name. The two files are automatically created during the specification of a fault injection campaign by the Loki front end, and each experiment is named at its beginning by the central daemon. Any node that is launched by the central daemon at the start of the experiment will be passed this information. The node invocation modification is only an issue if the system's nodes are dynamically started during an experiment.

The current implementation of the Loki runtime is in C. This requires that instrumented nodes be either written in C or interfaced with C.

2.2.5 Building a fault library

As previously mentioned, the responsibility for implementing the `injectFault()` methods for nodes requiring fault injections lies with the user. A convenient way to manage this responsibility is by building a fault library. One way to organize the fault library is by fault type. Each fault type is associated with a single injection method that takes a data structure of injection parameters as input. Each fault type is also associated with a translation method that translates a string of parameters for an injection into a data structure to be utilized by the injection method. A table can be built that resolves a fault type to its injection method and its parameter translation method.

The injection library can be utilized in several different ways. One is to statically set up the `injectFault()` method. In this technique, the `injectFault()` method resolves a fault name to a predefined injection method and injection parameter data structure. The downside to this approach is that the user's probe must be recompiled every time a fault is added or modified. A second approach is to embed the fault's type and injection parameters into its name. Here, a generic `injectFault()` method can be used. It simply retrieves the fault type and parameters from the fault name, and then resolves the fault type to an injection method and parameter translation method. The translation method can be used to translate the string of parameters from the fault's name into a data structure of parameters. This technique eliminates the first technique's problem of having to recompile the probe. However, it requires that the injection method and translation method be dereferenced and the parameters translated every time a fault is injected. The runtime could be modified to perform the dereferencing and translation during initialization, but that would require that this method always be used. A third approach is to use a fault definition file. Each fault is defined in the file with a fault type and injection parameters. When the probe is started it can read in the definition file and store the definitions in a table. Further, it can dereference the injection and translation methods, and translate injection parameters during the application's initialization. Therefore, it avoids excess overhead at injection time. Again, a generic `injectFault()` method can be used, eliminating the need to recompile the probe when faults are added or modified.

2.2.6 Campaign execution

Once the user has specified a campaign and its measures, and instrumented the system under study, the campaign is ready to be executed. Campaign execution is managed from the Experiment Manager (Figure 2.7), which is launched from the Loki Manager. It contains three panels. The top panel is responsible for execution parameters. The user assigns six parameters, which pertain to timestamps for use in clock synchronization. As previously mentioned, clock synchronization is performed during offline analysis to construct a global timeline for an experiment. The first two parameters indicate how many synchronization messages should be passed before and after either studies or experiments. The user also specifies the amount of time that should elapse between synchronization messages. Again, this is specified for both before and after messages. The next parameter determines whether "before" and "after" should refer to before and after each study or before and after each experiment. The final parameter is a port number on which the synchronization messages should be passed. For added convenience, these parameters can be saved for future use.

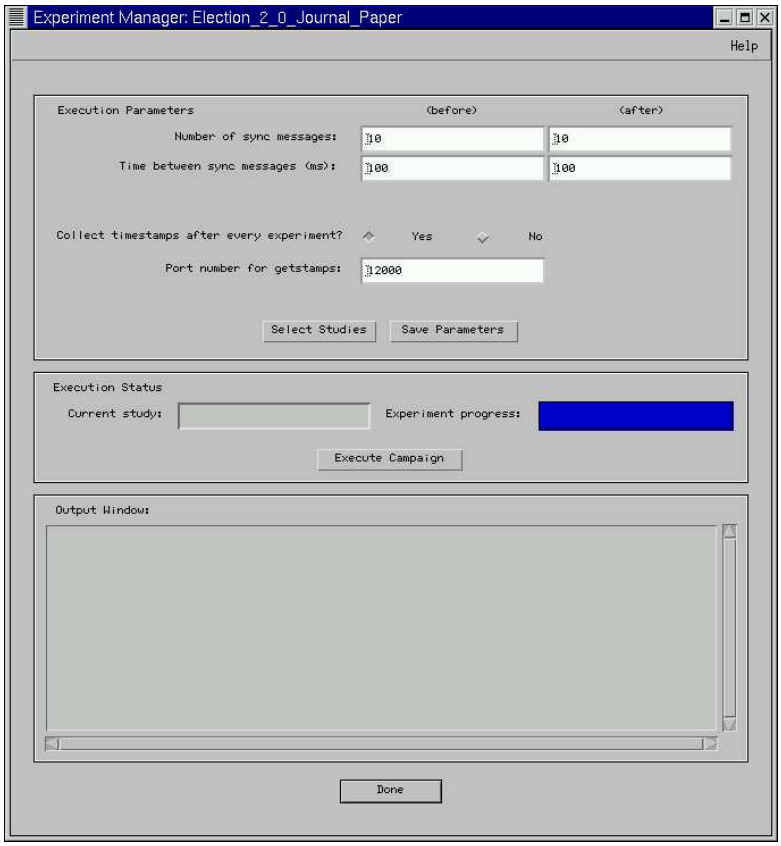


Figure 2.7 Experiment Manager

While the top panel is used during pre-execution, the middle panel tracks status during the campaign's execution. The final panel contains warning and error messages from the execution of a campaign.

Behind the scenes, the Experiment Manager oversees clock synchronization message passing and performs experiment coordination, daemon management, and experiment monitoring. The pseudocode in Algorithm 2.1 demonstrates the general algorithm that the Experiment Manager follows. In the code, *lokid* is used to refer to local daemons.

There are three phases to the CampaignExecution() algorithm. Each of the three phases is performed for each study from a selection of studies, before the algorithm moves onto the next study. In the first phase, the Experiment Manager (in the role of the central daemon) launches a local daemon on each host in the study's specification. Then the Experiment Manager waits for each local daemon to connect to it on a known socket and proceeds to establish a unique socket connection to each local daemon.

Following this local daemon initialization phase, the experiment execution phase begins. The experiment execution phase is performed on each experiment in the study. To start

```
CampaignExecution()
foreach _study_
   /* Local daemon initialization phase */
   openSocket() // Open socket for daemon connections
   foreach _host_
      exec(lokid)
      accept(remote_lokid_connection)

   /* Experiment execution phase */
   foreach _experiment_
      if (collect_timestamps_after_every_experiment = true ) or
      (_experiment_ = 0)
         foreach _host_
            exec(getstamps)
      foreach _lokid_
         send(start_experiment_msg, _lokid_)
      foreach _node_
         if requriesRuntimeStartup(_node_) = true
            send(start_node_msg, getlokid(_node_))
      while (experimentComplete() ≠ true ) and
      (time() < app_timeout)
         processDaemonMsgs(daemon_msg_buffer)
      if time() > app_timeout
         foreach _lokid_
            send(timeout_msg, _lokid_)
      foreach _lokid_
         receive(experiment_complete_msg, _lokid_)
      checkForExperimentErrors()

   /* Study completion phase */
   foreach _host_
      exec(getstamps)
   checkForStudyErrors()
   foreach _lokid_
      send(kill_msg, _lokid_)
```

Algorithm 2.1 Pseudocode for Campaign Execution in the Experiment Manager

an experiment, the Experiment Manager sends a message to each local daemon announcing the start of the experiment. The Experiment Manager also notifies the local daemons of nodes that they need to start. During an experiment, the Experiment Manager processes

messages from the local daemons. The Experiment Manager waits for the experiment to naturally complete or time out. If a timeout occurs, the Experiment Manager instructs the local daemons to shut down all nodes. The Experiment Manager then waits for experiment completion messages from each local daemon and checks for experiment errors. Also during the experiment execution phase, timestamps are passed for use in clock synchronization during postexperiment analysis. That happens either before the start of each experiment or just before the start of all experiments. As previously described, the user can change this option in the Experiment Manager.

After the experiment execution phase, the study completion phase is performed. It starts by gathering a final round of clock synchronization timestamps. Then the Experiment Manager checks for study errors and shuts down all local daemons.

Four aspects of the algorithm's implementation that deserve discussion are threads, remote process invocation, communication between the central daemon and local daemons, and experiment and study error detection. Threads are employed for various purposes by the Experiment Manager; it uses one thread to perform the CampaignExecution() algorithm, one thread to accept socket connections from local daemons, one SocketThread per local daemon to maintain a socket connection for communication, one thread per host to remotely execute the `getstamps`[5] process, and a graphics thread to refresh the Experiment Manager display.

In addition to threads, the Experiment Manager makes extensive use of remote process invocation for executables such as `getstamps` and `lokid`. It does so via the SSH secure shell client. The stdout and stderr I/O files of remote processes are redirected to appropriate log files to be examined by the front end. Although stdout and stderr I/O files are useful for logging purposes, they are not adequate for all communication required by the front end. In particular, TCP sockets are used for communication between the central daemon and the local daemons. On the central daemon side (the Loki front end), each socket connection is maintained in a separate thread, called a SocketThread. When messages arrive from the local daemons, they are tagged with identifying information and queued in a daemon message buffer. The message buffer has a lock for concurrent access. The CampaignExecution thread handles these messages as designated in Algorithm 2.1. Each SocketThread also maintains a socket lock for its outgoing message buffer.

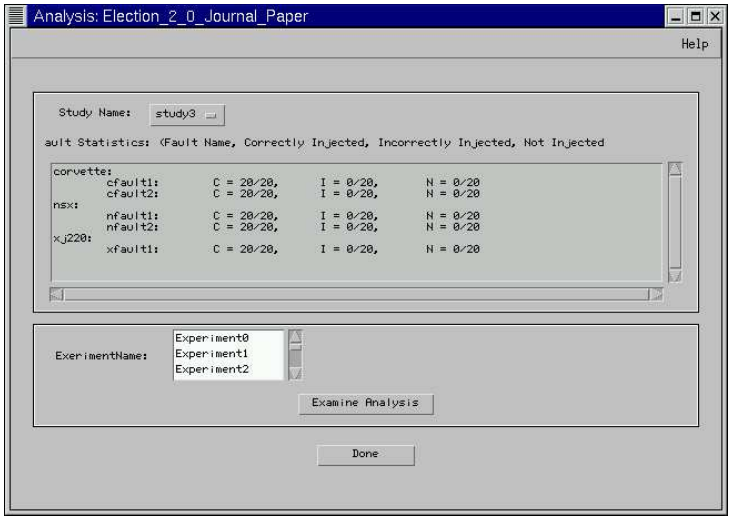
Throughout the campaign execution process, the Experiment Manager must monitor for errors. It does so by examining the log files created by remote processes, the messages received on sockets, and the status of socket connections. It handles errors by archiving

[5] `getstamps` is the utility that Loki uses to generate and collect clock synchronization messages.

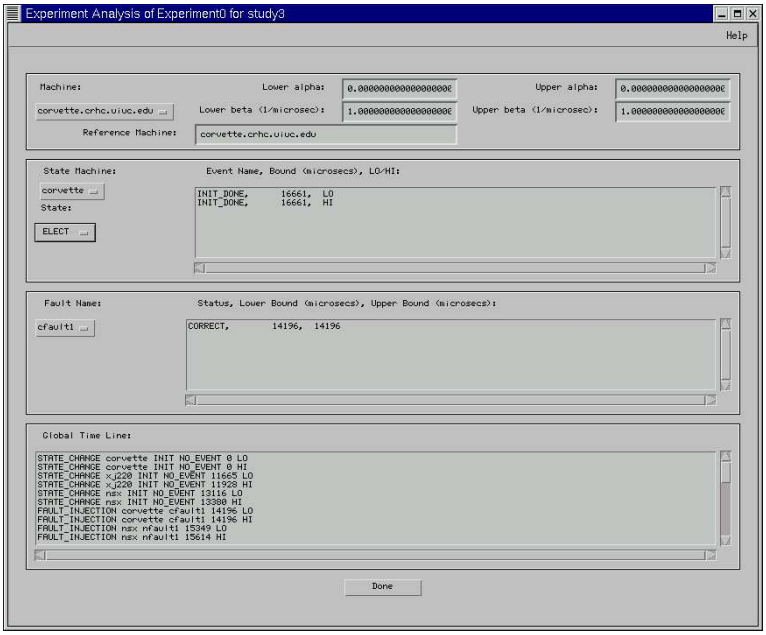
failed experiments for later examination.

2.2.7 Offline processing

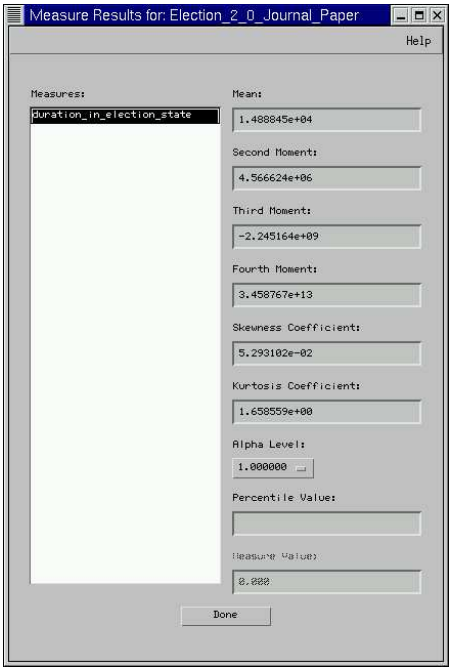
After the user has finished campaign execution, the campaign is ready to be processed offline by Loki. As described earlier, there are two phases to the offline processing: experiment analysis and measure computation. Each step is invoked from the Loki Manager. Experiment analysis performs the analysis phase of Loki, as described in [14, 18], to create global timelines and determine the correctness of fault injections from the raw data collected during the campaign's execution. The Experiment Analysis Window, pictured in Figure 2.8(b), displays several details of an analyzed experiment, such as the α and β values for the clock synchronization, the timing information on when state machines entered particular states, the status (correct injection, incorrect injection, or not injected) and timing information for faults, and the global timeline for the experiment. After the analysis is performed, the user presses the "Compute Measures" button in the Loki Manager to carry out measure computation (as described in Section 2.1.4). Once that has been done, the observation and measure executables that were compiled during the measure specification phase are run in order to calculate statistical properties of the desired campaign measures. They are displayed in the Measure Results Window, seen in Figure 2.8(c). The properties include the first four central moments, the skewness coefficient, the kurtosis coefficient, and approximate percentile values.



(a) Analysis Summary Window



(b) Experiment Analysis Window



(c) Measure Results Window

Figure 2.8 Loki Offline Processing Screen Shots

CHAPTER 3

A LIGHTWEIGHT FAULT INJECTION PROXY FOR LOKI

As demonstrated in previous chapters, Loki is a sophisticated tool for evaluating distributed systems. Its global state-triggering mechanism, separation of the fault injection mechanism from the fault injection policy, and measures support make it a good choice for many studies. An example is the experimental study of the Ensemble group membership protocol [17]. However, with any tool, the situations in which the tool can be applied are dramatically affected by design and implementation decisions. While we were attempting to use Loki to perform the fault injection case study of the Coda distributed file system, it soon became apparent that the existing Loki runtime, described in Section 2.1.3, would not be adequate. In particular, instrumentation of the Coda file server with Loki proved to be a daunting task. It was possible to compile a Coda file server with the Loki runtime; however, the file server would not properly initialize when run with Loki, causing the file server to hang. Through debugging, the problem was isolated to a call that never returned in Coda's lightweight process (LWP) package. This lead to the conclusion that there was a conflict between Coda's LWP package and the POSIX threads (Pthreads) model that Loki uses. Thus, we either needed to resolve the problem between the two threading packages, or we had to find an alternative way to use Loki. Resolving the conflicts in the threading packages would have been an arduous task that might have had no clear solution. Therefore, we pursued the second option.

We developed the idea of using a lightweight fault injection proxy. Instead of integrating the runtime into an application node directly, we integrate it into a *proxy* process that services Loki runtime functionality for the node. This allows us to integrate the application with a much more lightweight Loki component. The only purpose of this lightweight component is to provide the application with a means to notify Loki of local events, so Loki can track the

state of the application. This component is referred to as the *event mediator*. The advantage of the proxy approach is that many of the heavyweight functions of the Loki runtime have been offloaded to a separate process, reducing the burden on the application under study. The proxy solution is not specific to the above example of threading package conflicts. The approach can be taken advantage of in a number of situations in which one wishes to mitigate potential conflicts between an application and Loki. The disadvantage is that this method adds a level of indirection for some of the Loki runtime services and may add undesired latencies. It certainly adds a level of indirection for event notifications. It also limits the types of faults that can be injected, because faults are now injected from the proxy rather than within the application.

Another motivation that shaped the design of the proxy was the need for a way to drive experiment execution. Most distributed systems involve behavior based on responses to internal and external input to the system. This input may include, for example, client requests, messages, or system events. In order to study system behavior, an evaluator must have some mechanism for guiding and driving the state of the system. Along the same lines, some experiments require certain setup and cleanup actions. All of these factors were considered and integrated into the design of the proxy.

3.1 Experiment Control Features

The proxy takes advantage of the existing Loki framework to allow several experiment control features. Some of the features are implemented within the proxy, while others represent semantic changes in the way the original Loki architecture is used. An important semantic change that the proxy takes advantage of is that of using fault injections as stimuli to the system under study, providing the experimenter with a way to guide the state of the system. Instead of creating a new mechanism to stimulate the system, the proxy takes advantage of the fault injection mechanism already in place. The existing mechanism not only provides a means to stimulate the system, but also provides temporal control based on global state triggering. The stimuli will be called *control modifiers*. Although control modifiers use the same injection mechanism as faults, many injections still obey the traditional notion of a fault as a potential cause of an error, and are referred to simply as *faults*. When we refer to the fault injection mechanism, fault definitions, or any other general notion that applies to faults, the same notion applying to faults will also apply to control modifiers, because they both use the same injection mechanism.

Another control feature is the *auxiliary event*. In contrast to control modifiers, auxil-

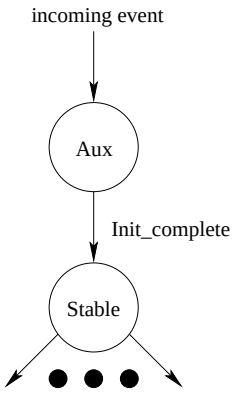


Figure 3.1 Example of an Auxiliary State

iary events require implementation support rather than semantic changes. There are three types of auxiliary event. All three types are generated upon completion of a fault injection. The first type notifies the state machine that the associated fault injection has completed, and is called a *complete event*. Complete events are named according to the form <fault name>_complete. Closely related to complete events are *returnval events*. They embed a numeric return value from a fault injection into the event's name. The format for the event name is <fault name>_ret_<return value>. Complete and returnval events are important because an experimenter can build state machines based on knowledge about faults. This capability is useful for a number of reasons. For example, the user may have a fault that is not instantaneous and is ongoing while the global state is changing. The user may condition part of his/her state machine based on the fault's completion. Also, the user may want to track the return status of a fault to see if it performed correctly. Finally, the third type of auxiliary event is the *globally conditioned event*. Events of that type are named according to the form <fault name>_globalcond, and indicate the occurrence of global conditions in a state machine; thus, the associated faults do nothing.

The final experiment control feature is the *auxiliary state*. Normally, states represent knowledge concerning the system under study. Auxiliary states, on the other hand, represent knowledge that is beyond the system under study, such as experiment state. As with control modifiers, the support of auxiliary states involves semantic changes to the original design for Loki. There are three ways that auxiliary states have been employed so far. The first way is to perform synchronization within a state machine. In Figure 3.1, the auxiliary state (`Aux`) could be used as the trigger state to perform an initialization procedure in the system under study. The `incoming event` could be the start of the experiment, an auxiliary event, or some other system event. When the state machine enters the `Aux` state, an Init control modifier is triggered to perform the initialization procedure. Upon its completion, a complete event

could be used to signal to the state machine that the initialization has completed, in which case the state machine transitions to the `Stable` state. From the `Stable` state, a fault could be injected. In this way, the initialization procedure and the fault have been ordered in the experiment. Furthermore, it is known that the initialization procedure completely finished before the fault was injected. The second way an auxiliary state has been employed is to perform synchronization between state machines. Again, consider the example in Figure 3.1. This time the Init control modifier is triggered when two separate nodes both enter the `Aux` state. An `Init_complete` event is still created at the end of the Init control modifier. In this way, the two state machines can be synchronized at the `Stable` state after each node has performed the initialization procedure. The third way an auxiliary state has been employed is as a *sanity check state*. When multiple nodes have synchronized at a sanity check state, a sanity check is invoked through the control modifier interface. When the sanity check is through, a returnval event is created, returning the status of the sanity check. This is useful for a number of reasons. For instance, some initialization and setup may need to be performed at the beginning of an experiment. If the initialization and/or setup fails, it is detected by the sanity check. The experiment could then be aborted.

The addition of control modifiers, auxiliary events, and auxiliary states provides the proxy with a means for experiment control. State machines no longer just represent the status of the system under study. They now combine the status of the experiment with the status of the system under study.

3.2 Proxy Design

Having discussed the major features and motivation for the proxy, we will now present the design details. We will include a description of the new look of the Loki architecture with the proxy incorporated, and a discussion of how faults are handled.

3.2.1 The proxy architecture

The proxy design introduces to the Loki architecture two new software components, the proxy and the event mediator, as mentioned earlier. The event mediator is implemented in a library called `libappproxy.a`. A node in the system under study is instrumented with the event mediator instead of with a Loki runtime as in the classic Loki architecture. The Loki runtime is offloaded to the proxy process `lokiappproxy`, which performs Loki runtime functionality on behalf of the system node. The new architecture is illustrated in Figure 3.2. Communication is one-way, traveling from the node to the proxy. The event mediator is

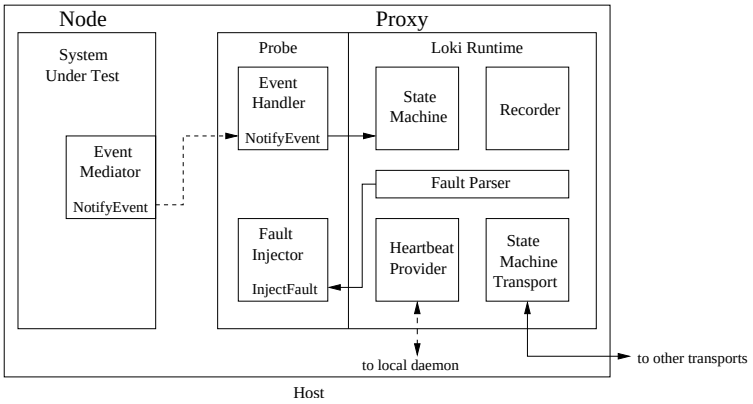


Figure 3.2 Lightweight Proxy Architecture

used to notify the proxy when local events occur. These events are handled by the event handler inside the proxy runtime and passed off to the Loki runtime. When the fault parser determines that a fault should be injected, the proxy's fault injector handles the injection. All Loki runtime functionality performs as described in Section 2.1.3.

The major benefit of this design is that the Loki component that is integrated in the system under study is considerably more lightweight than the component that the original Loki runtime required. For the Coda case study discussed in this thesis, the proxy design has effectively removed Loki's need for thread support within the system under study, thus eliminating potential conflicts in threading packages. All that is required of the system under study is to make a function call when local events occur. The function call simply communicates with the proxy to let it know about the occurrence.

It is necessary to offload all of the Loki runtime functionality, other than local event notifications, to the proxy in order to eliminate the need for thread support within the application under study. The state machine transport requires thread support in order to communicate asynchronously with other state machine transports, and cannot be left in the system under study. The state machine requires asynchronous access to state data, so the state machine must also be offloaded. The fault parser, and consequently fault injection, must be moved out of the system under study too, because they require asynchronous notification of state changes from the state machine. Similarly, the recorder requires thread support, because it is asynchronously accessed when fault injections occur. Finally, the heartbeat provider requires thread support to generate heartbeats at regular time intervals and is also moved to the proxy.

3.2.2 How faults are handled

An important implication of the proxy design is that faults can no longer be injected from within the application under study. This unfortunately limits the types of faults that can be injected; however, there are still several types that can be injected. In the proxy, faults are divided into two categories: external faults and internal faults. External faults are ones in which the proxy creates a new process to execute them. External faults include any actions that are performed in their own executables. For example, an external fault may be a script that modifies a resource external to the proxy and the system under study, or a script that modifies the system under study through the system's own external interface. Internal faults are faults that are performed using function calls within the proxy. An example would be a system call that alters global resources to which the system under study requires access. As discussed in Section 2.2.5, a fault library is a practical way to manage faults. Internal faults are managed with a fault library. Each type of internal fault is associated with an injection method that takes a data structure of injection parameters as its input. Each of these injection methods is also associated with a translation method that translates a string of parameters into an appropriate data structure to be passed to the injection method.

Section 2.2.5 also mentioned that there are several ways to employ the fault library. The proxy utilizes the fault definition file approach. The format for fault definitions in the file is {e|i}{n|c|r|g}:<fault name>=<fault parameters>. The first option is an e or an i, indicating whether the fault is external or internal. The second option indicates the type of auxiliary event that should be generated, if one is to be generated. n indicates that an auxiliary should not be generated. c, r, and g indicate respectively that a complete, returnval, or globally conditioned event should be generated. The fault name is the same name that is specified in the node's fault specification. There are a few options for the fault parameters. If the fault is an external fault, then the fault parameters are a command line to be executed outside of the proxy. If the fault is an internal fault, then the fault parameters are an internal fault type followed by parameters for the fault. Each internal fault type has its own parameter structure. The final fault parameter option is the specification of NO_FAULT. This is used to generate an auxiliary event without generating a fault or control modifier. An example of a fault definition is in:terminate=CRASH now. This indicates that there is an internal fault of type CRASH named terminate that does not result in an auxiliary event, and the parameter to the fault is now.

In addition to fault definitions, the fault definition file contains <tag>=<value> pairs to specify certain parameters to the proxy. The following tags are reserved for this purpose: PRE_APP_EXE, POST_APP_EXE, G_SHUTDOWN_CMD, and G_SLEEP_TIME. None of these reserved

tags need to be used in the fault definition file. Instead, they are available if needed for a study. The `PRE_APP_EXE` and `POST_APP_EXE` tags are used to specify executables to be run at the beginning and end of an experiment, respectively. The intention of these executables is that they allow a setup script to be performed before an experiment and a cleanup script to be performed at its conclusion. The `G_SHUTDOWN_CMD` tag is used to specify a command to gracefully shut down the system under study should it be terminated by the proxy. This may happen, if for instance, an experiment times out. Some systems may require a proper shutdown to eliminate residual effects in the next experiment. Lastly, the `G_SLEEP_TIME` tag is used to specify how long a graceful shutdown command should be allowed to execute.

The fault definition file is read and processed before an experiment even starts, improving the performance of fault injections. The algorithm that is used for fault injections is also an important part of the proxy design. It is described with pseudocode in Algorithms 3.1 and 3.2.

```
InjectFault(_fault_name_)
_fault_ ← resolveFault(_fault_name_) // Determine all fault details
if generateAuxiliaryEvent(_fault_) = true
   enqueue(_fault_, _pending_fault_queue_)
   _time_ = time()
else
   _time_ = InjectFaultHelper()
return _time_
```

Algorithm 3.1 Pseudocode for Fault Injections

```
InjectFaultHelper(_fault_)
switch faultType(_fault_)
case EXTERNAL
   exec(_fault_)
case INTERNAL
   performFault(_fault_, _fault_library_)
case NO_FAULT
   // Do nothing
_time_ = time()
if generateAuxiliaryEvent(_fault_) = true
   notifyEvent(auxiliaryEvent(_fault_, _time_))
return _time_
```

Algorithm 3.2 Pseudocode for the Fault Injection Helper Routine

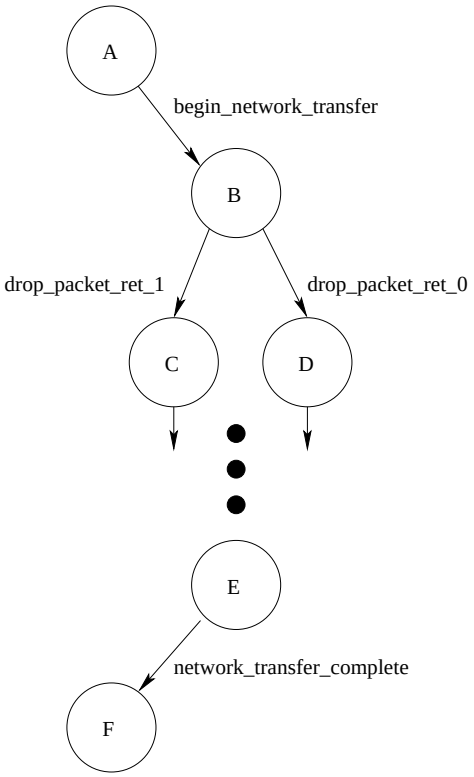


Figure 3.3 Example of a State Machine Supporting Pending Faults

There are several points to be noted in the fault injection algorithm. First, we need to consider the relationship between the time a fault is injected and the time it is handed to the InjectFault() method. If the fault generates an auxiliary event, the InjectFault() method places the fault in a queue of pending faults, rather than immediately injecting the fault. The queue is handled by a collection of fault injection threads that use the InjectFaultHelper() when they inject pending faults. This queue allows multiple faults generating auxiliary events to remain pending. This may be necessary for several reasons; Figure 3.3 illustrates one scenario. In it, a network_transfer control modifier that generates a complete event is started from state `A` but does not finish until state `E`. During its execution, the fault drop_packet is injected during state `B`. The fault generates a returnval event. A return value of 0 indicates that a control packet was dropped, and a value of 1 indicates that a data packet was dropped. Without the pending fault buffer, the network_transfer control modifier and drop_packet fault would not both be able to be outstanding at the same time, and this scenario would not be possible. If no auxiliary event is to be generated, then the fault is injected immediately, and InjectFault() does not return until the fault is completed.

This brings up the timestamp issue. If a fault generates an auxiliary event, then its timestamp is the time that it was added to the pending fault queue. This is fine, because an

auxiliary event will later be generated from the InjectFaultHelper() that will be timestamped with the completion time of the fault. If the fault does not generate an auxiliary event, then the time just following the completion of the fault is recorded. The final aspect of the fault injection algorithm to be noted is the handling of different types of faults. External faults are executed outside the proxy process, while internal faults are executed through a function call within the proxy. There is also a provision for no fault to occur during an injection, as previously mentioned.

3.3 Proxy Implementation

Several details of the proxy implementation outside the scope of its design deserve discussion. Implementation details have an impact on the viability of the proxy as an alternative approach to using Loki. Because the proxy is instrumented with a Loki runtime, the implementation of the proxy faces all of the instrumentation issues discussed in Section 2.2.4.

In addition, an important consideration is the communication between the proxy and the event mediator. Communication is one-way from the proxy to the event mediator, to eliminate thread support for handling incoming messages at the event mediator. Since a node's proxy is located on the same host as the node, communication is performed using shared memory and semaphores. This requires that a small initialization procedure be used at the startup of the node to set up communication from the event mediator to the proxy.

As previously mentioned, threads have been eliminated from the event mediator, but they are still used within the proxy. In addition to the threads that the Loki runtime requires when included in the proxy, the proxy makes use of threads to support event notifications and fault injections. The proxy uses one thread to read event notifications that were sent from the event mediator. It also uses a collection of threads to handle fault injections that were placed in the pending faults buffer as described in Section 3.2.2.

Another important issue that the proxy faces is the proper implementation of signal handlers. Since the proxy is acting on behalf of a node in the system under study, it must handle signals from the Loki runtime for itself and for the node. The first thing that the proxy does when it receives a signal is to perform a minimum cleanup routine. This routine identifies the signal and passes an appropriate signal to the proxy's node. It then cleans up the communication channel between it and the event mediator. Finally, the Loki runtime signal handler is called so it can perform appropriate cleanup actions. An important consequence is that the application under study is forced to handle a signal. Since signals must be promptly handled, the proxy does not take the time to shut the node down

gracefully. However, there are times when a graceful shut down is appropriate. Graceful shutdowns are supported by allowing the user to specify an external command in the fault definition file that can be used by a proxy to shut down a node from the system under study. A graceful shutdown is used if there is a timeout on an experiment and if the proxy prematurely shuts down due to unexpected errors during execution. Graceful shutdowns are application-specific, but can be very useful, especially when an experiment times out. When that happens, the system under study will be given ample time to clean up so that the next experiment can run properly. The introduction of a graceful shutdown requires that the user be able to specify a graceful shutdown sleep time in the fault definition file. This is the amount of time that the proxy allows for the performance of the graceful shutdown command before continuing its execution.

In addition to these proxy implementation details, some small implementation changes were also made to the Loki runtime. The first change was to the daemon. It was modified to accept an experiment complete timeout value that the user specifies when designing a fault injection campaign. It is the amount of time that the daemon should give the proxy to shut down after the daemon has asked it to. It is particularly important now that the graceful shutdown has been supported because a proxy may require a nontrivial amount of time to perform the shutdown. The second change was the addition of support for the graceful shutdown in the Loki runtime cleanup routine. This was needed so that if the daemon asks the proxy's Loki runtime to shut down, the graceful shutdown command will be performed on the proxy's node. The final change was the addition of support to the parser so that event notifications could arrive during a fault injection. This support is important because it allows a fault to execute for a nontrivial amount of time. We added the support by slightly changing the parsing algorithm. Algorithm 3.3 shows the execution of the new parser algorithm. Lines 2 through 4 were added to provide the new support. When a new event notification is processed, the $\_pending\_state\_change\_$ variable is set to true, and the $\_notify\_event\_condition\_$ is signaled. If the parser is waiting in line 3, it will be awakened when the $\_notify\_event\_condition\_$ is signaled, and it will continue parsing to see if any faults should be injected. If an event notification occurs while the parser is anywhere else in the algorithm, the parser will not miss the new state change, because it checks the pending state changes before waiting on the $\_notify\_event\_notification\_$.

```
PARSER()
(1)   while (1)
(2)     if _pending_state_change_ = false
(3)       WAIT(_notify_event_condition_)
(4)     _pending_state_change_ ← false
(5)     foreach _fault_
(6)       LOCK(_state_table_)
(7)       _fault_expression_ ← GETFAULTEXPRESSION(_fault_)
(8)       _previous_value_
(9)         ← GETPREVIOUSFAULTEXPRESSIONVALUE(_fault_)
(10)      _injection_ready_
(11)        ← PARSE(_state_table_, _fault_expression_, _previous_value_)
(12)      UNLOCK(_state_table_)
(13)      if _injection_ready_ = true
(14)        INJECTFAULT(_fault_)
(15)
```

Algorithm 3.3 Pseudocode for Parser Execution

3.4 Using the Proxy

Use of the proxy requires a campaign specification, as all Loki fault injection campaigns do. Furthermore, fault definition files must be created with the format described in Section 3.2.2 for each node in the system under study. To use the proxy, it is necessary to make a minor change in the campaign specification. It is also necessary to modify the node's application and application arguments that were specified from the State Machine Editor. The specified application should be changed to `lokiappproxy`, the executable name for the proxy. Its arguments should be set to the location of the node's fault definition file, followed by the original application and its arguments. In addition, it is necessary to perform the following steps to instrument an application for use with the proxy. They are similar to the standard instrumentation steps required for use of the Loki runtime, but they reflect the integration of the Loki runtime into the proxy. The proxy code discussed below is supported in the `libappproxy.a` library.

1. *Initialization procedure:* The first instrumentation step is to place the proxy's initialization procedure as the first code that is executed in the `main()` method of the node that the proxy is representing. The initialization procedure is contained in the C macro `INITIALIZE_PROXY_COMM()`. The macro takes the application's command line arguments as its input. The macro is needed to set up communication between the node and the proxy.

2. *Event notification:* The next step is to use the `lokiNotifyEvent()` method to notify the proxy when local events occur that could trigger transitions in the node's state machine. The parameters to the method are the name of the event and the time at which it occurred.

3. *Fault implementation:* The proxy already has the fault injection method implemented, but in order for the internal faults to be included in the proxy's fault library, the proxy requires the implementation of an injection and translation method.

4. *Node invocation:* The final step is to modify node invocations within the application under study. If a proxy is to be used for a node, then the node invocation should be changed to reflect the following format: `lokiappproxy <study file> <daemon contact file> <experiment name> <fault definition file> <application> <application arguments>`. The study file, daemon contact file, and experiment name are generated by Loki and are required, as they are for standard Loki runtime instrumentation. The fault definition file is the one described in Section 3.2.2. The application and its arguments are associated with the node on whose behalf the proxy is acting. As with instrumentation using the standard Loki runtime, this step is only required if nodes are dynamically started in the system under study.

3.5 Network Fault Injector

The final elements of the proxy we will discuss are the internal fault injectors that have been built for its fault library. We have created a versatile network fault injector that supports the denial of packets into and out of a host.

Our first attempts at supporting a network fault injector made use of external interfaces to create network partitions. The first external interface was a Coda-specific filtering tool. It filtered network traffic that used Coda's RPC2 protocol. The advantage of this approach was that the filtering tool could be easily used through the proxy's external fault interface. The disadvantages were that it was Coda-specific (which is not important in the context of our Coda case study) and, more importantly, that it took too long to create the desired effects. This second disadvantage was due in large part to the fact that the fault injections were performed outside of the proxy, thus requiring that a new process be created and destroyed before the injection was complete. This latency proved to be too great for the Coda case study, and a new network fault injector had to be created. The second attempt involved use of external access to IP Chains, a rule-based interface to IP firewall administration in the

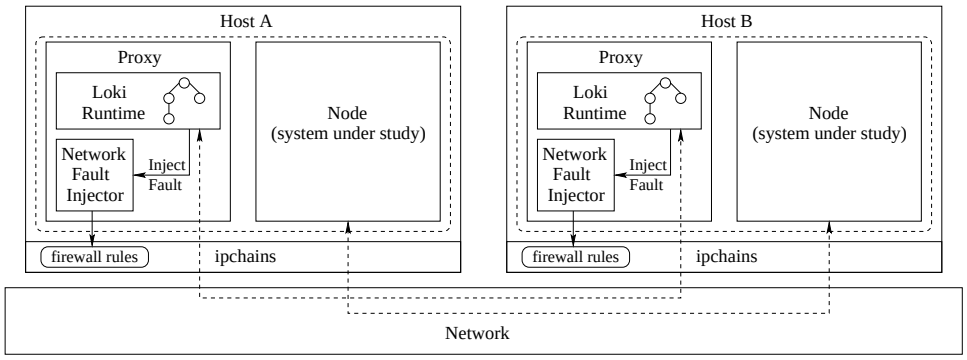


Figure 3.4 Network Fault Injector

Linux kernel. The advantage of this approach was that it too could be easily accessed through the proxy's external fault interface. One disadvantage to this approach was that depending on the configuration of the host on which the proxy ran, it would likely be necessary to run the proxy with operating system administrative privileges. The other disadvantage was that it also proved to take too long to produce the desired effects. Thus, we concluded that an internal injection approach would be necessary.

Fortunately, underneath the IP Chains external interface, there are system calls that could be invoked from within the proxy. Not only did this method drastically improve injection efficiency, but it also helped reduce intrusiveness by eliminating the need for a new process when the fault injector was used. Therefore, this method was chosen. Figure 3.4 illustrates how the network fault injector fits into the Loki architecture.

The following is the interface for the network fault injector. Each command in the interface is specified with the parameter list that it requires in the fault definition file and a description of its effects. A host in the parameter list can be denoted by a host name or an IP address. The two port numbers specify a range of ports to which the command is applied. The protocol refers to a network transmission protocol and is specified with a number, as defined by the operating system. For Linux, UDP is 17 and TCP is 6. Also, a given network fault can utilize more than one command by listing the commands' parameters one after another in its definition in the fault definition file.

- `denyAllIn <port0> <port1> <protocol>`: This command denies all network traffic arriving at a port in the indicated range using the specified transmission protocol.
- `delDenyAllIn <port0> <port1> <protocol>`: This command disables a `denyAllIn` command with the same parameters.
- `deny1In <host> <port0> <port1> <protocol>`: This command denies network traf-

fic arriving from the specified host at a port in the indicated range using the specified transmission protocol.

- `delDeny1In <host> <port0> <port1> <protocol>`: This command disables a `deny1In` command with the same parameters.
- `denyAllOut <port0> <port1> <protocol>`: This command denies all network traffic leaving from a port in the indicated range using the specified transmission protocol.
- `delDenyAllOut <port0> <port1> <protocol>`: This command disables a `denyAllOut` command with the same parameters.
- `deny1Out <host> <port0> <port1> <protocol>`: This command denies network traffic leaving for the specified host at a port in the indicated range using the specified transmission protocol.
- `delDeny1Out <host> <port0> <port1> <protocol>`: This command disables a `deny1Out` command with the same parameters.
- `flushIpChains`: This command flushes all firewall rules loaded by IP Chains.

CHAPTER 4

CODA CASE STUDY OVERVIEW

This chapter presents an overview of the Coda fault injection case study. It starts with an overview of Coda, followed by the design of the case study.

4.1 Coda Overview

4.1.1 Review of Coda concepts

Coda [5] is a distributed file system designed to be highly available. It tolerates both network partitions and server crashes through server replication and disconnected operation. *Server replication* allows data to be replicated across multiple servers and makes the data accessible to a client even if not all of the servers are reachable. In Coda, data is replicated at the granularity of a *volume*. The data to be stored in the file system is divided into volumes, such that a subdirectory of the file system and the subdirectory's contents form a volume. The servers that host a replica of a particular volume form a *volume storage group* (VSG), and the subset of this group that is reachable by a client is the client's *available volume storage group* (AVSG). *Disconnected operation* allows a Coda client to cache data locally. The resulting local repository is used if the client cannot reach any servers from a volume's VSG. Replication of data in the file system is made transparent to the user whenever possible.

In conjunction with these mechanisms, Coda uses an *optimistic data consistency* scheme. The optimistic consistency scheme allows partitioned reads and writes to occur by making an optimistic assumption that the same data object will not be accessed in separate partitions. Although the optimistic scheme improves the availability of data in the file system, it requires that the file system support a mechanism for handling diverging replicas, in case the optimistic assumption does not hold. This mechanism is called *resolution*, which is more

formally defined in [6, p. 13] as "the process of converging diverging replicas to the same value." The burden of detecting replica divergence is left to *Venus*, Coda's client cache management process. When a client accesses a replica and Venus detects a divergence, Venus requests that its AVSG resolve the divergence.

Coda breaks down resolution in two ways: automatic versus manual resolution, and directory versus file resolution. *Automatic resolution* is resolution that Coda can perform by itself without ambiguity; divergences that cannot be automatically resolved require that a human perform *manual resolution*. A distinction is made between *directory resolution* and *file resolution* because directories have well-defined semantics that allow many divergences to be resolved automatically, while files do not. In the rest of this paper, the terms *directory resolution* and *file resolution* are used to refer to the resolution of a directory object and of a file object, respectively, and *resolution* or *volume resolution* is used for the process of resolving an entire volume, which is really just a series of directory and file resolutions.

4.1.2 Description of the directory resolution protocol

This subsection presents a brief description of Coda's directory resolution protocol. The protocol is considered for a volume that consists of exactly one directory, since that is what is used in the case study's experiments. The protocol is started when a Venus process sends a ViceResolve request to a server, making the server the *coordinator* of the directory resolution. The remaining servers in the client's AVSG become *subordinates* in the protocol. Together the coordinator and the subordinates form a *directory resolution group* (DRG). The directory resolution protocol consists of several phases that are performed one after another as requested by the coordinator. Subordinates are unaware that the protocol is taking place. They simply fulfill requests as they are made by the coordinator. During Phase 1 (LockVol) of directory resolution, the coordinator determines which servers are available to participate in the protocol and asks them to lock the volume containing the directory to be resolved. It then compares the statuses of the directories at each server in the DRG to determine if resolution is needed and computes a global transitive closure, i.e., the set of directories that must be resolved for the current directory resolution request. For this case study, the global transitive closure consists only of the directory for which resolution is requested. More details about global transitive closures can be obtained from [6]. In Phase 2 (FetchLogs), the coordinator collects from subordinates all logs that it needs to resolve the directory and merges the logs into a single buffer. Then, in Phase 3 (ParseLogs), the coordinator distributes the combined log buffer to the subordinates, which, along with the coordinator, perform compensating operations to bring their copies of the directory up to date. The

subordinates return any inconsistencies that arise to the coordinator. If inconsistencies are reported, Phase 3.5 (HandleInc) is used by the coordinator to distribute the list of inconsistencies to subordinates to verify that the inconsistencies exist. If no inconsistencies exist, then in Phase 4 (InstallVV) the coordinator ships a new storeid for the directory to each subordinate, and the DRG commits the updated directory. If inconsistencies do exist, Phase 4 is skipped and Phase 5 (MarkInc) is used by the coordinator to order the DRG to mark the volume inconsistent. Finally, Phase U (UnlockVol) is used by the coordinator to instruct the DRG to unlock the volume.

4.2 Case Study Design

Having reviewed Coda, the design of the fault injection case study is now presented. The case study constitutes a single Loki fault injection campaign. Section 4.2.1 describes the experiment scenario and breaks it into studies for the campaign. Then in Section 4.2.2, we describe the Loki state machines, faults, and control modifiers that are deployed as input to Loki during experiments for those studies. We conclude our overview of the case study design with a discussion of Coda instrumentation details.

4.2.1 Experiment scenario and division of Loki studies

The Coda case study focuses on studying the effects of correlated network partitions on the resolution process. In particular, it considers correlated network partitions that occur during a phase of the directory resolution protocol. To study them, we designed an experiment with two Coda servers, each on its own host. The servers were each limited to one LWP (lightweight process) to serialize the work performed at that server. A Coda client was also started on each of the hosts. After the servers and clients were initialized, a volume was replicated across the two servers and initialized with data. The volume, called the *target volume*, consisted of a single directory containing several files. After that setup was performed, a network partition was injected. During the partition, each client read from and wrote to its local replica. That workload was designed such that the resolution of the partitioned updates could be performed automatically. After the clients were done with their workloads, the network partition was repaired. One of the clients then requested that the entire target volume be resolved. While the servers were performing the resolution process, a correlated network partition was injected during some phase of the directory resolution protocol to resolve the target volume's directory. That caused the resolution process to abort. After it aborted, the network partition was once again repaired, and resolution was again

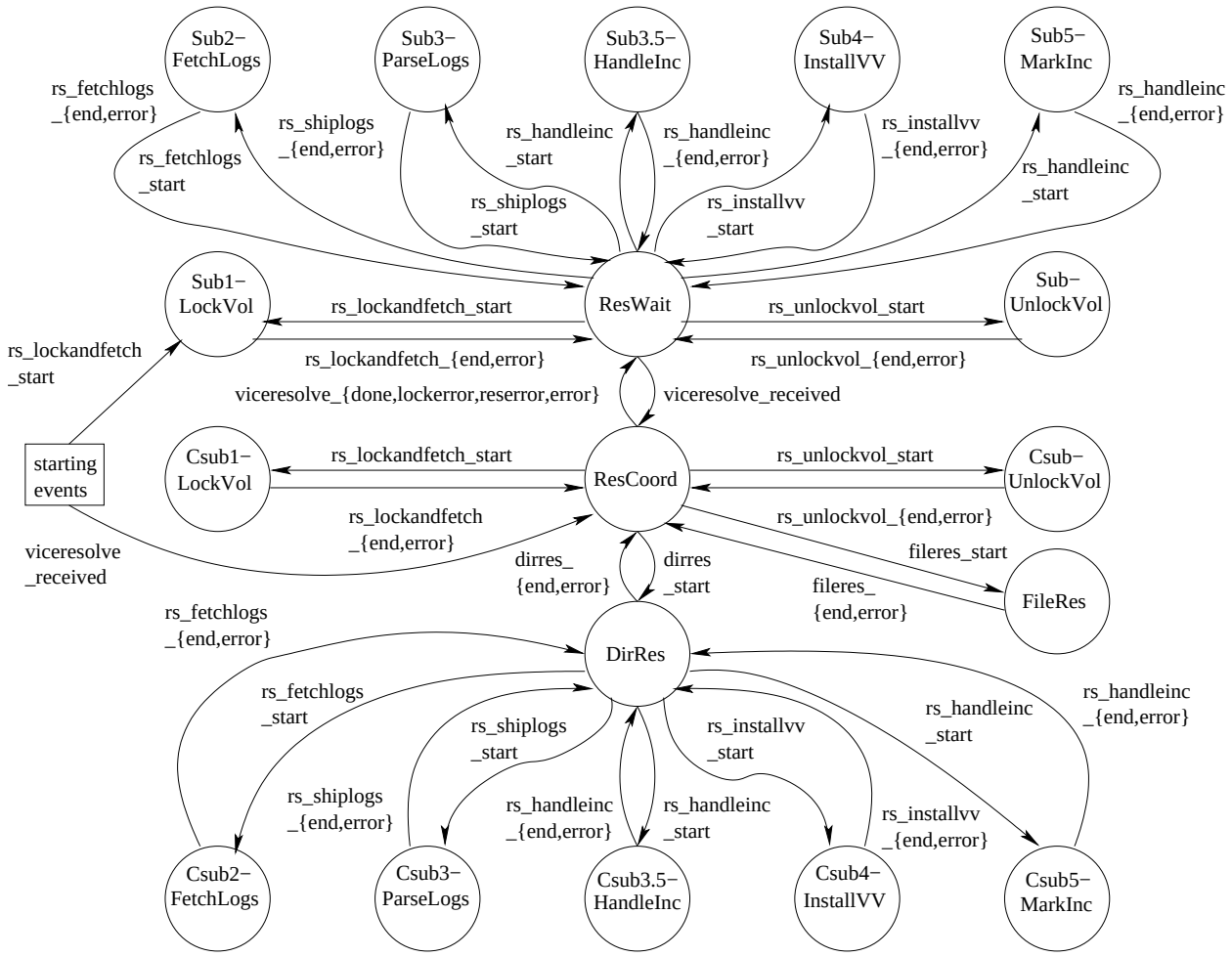


Figure 4.1 Coda Resolution State Machine

requested. This time, the resolution process was allowed to finish. Correlated injections have been performed in Phases 3 and 4 of directory resolution. Loki fault injection studies pertaining to those scenarios are referred to as *studies P3* and *P4*. In addition, a baseline case, called *study B*, was run without the correlated network partition.

4.2.2 Loki state machines

4.2.2.1 Coda resolution state machine

The Loki state machine in Figure 4.1 was constructed to represent the state of a Coda server during the resolution of the target volume. This state machine is used as input to Loki, allowing Loki to track state, which is it needs to do in order to trigger fault injections and to compute measures. In the state machine, vertices represent states and arcs represent event-triggered transitions. If the symbols { and } appear in the name of an arc, it means

that more than one event can cause the transition. For instance, `viceresolve_{end,error}` represents both the `viceresolve_end` and `viceresolve_error` events. Also, some arcs are labeled with comma-separated lists of events that can cause the associated transitions. The box labeled "starting events" at the left side of the state machine is where a node enters the state machine.

To invoke resolution, for each object in a volume that it wishes to access, a Venus process sends one ViceResolve request to a server in its AVSG. As mentioned above, that server becomes the coordinator of the resolution request, and the remaining servers in the AVSG become subordinates. A `viceresolve_received` event represents this action and causes the coordinator to arrive at the `ResCoord` state. The `ResCoord` state is the home state of the coordinator between resolution actions. From it, the coordinator performs either directory or file resolution, depending on the request made by the Venus process. Either way, it first coordinates with the subordinates to lock the diverging volume. When a volume lock request arrives from the coordinator, each subordinate generates an `rs_lockandfetch_start` event that transitions the subordinate to the `Sub1-LockVol` state. When a subordinate is finished locking the volume, it generates an `rs_lockandfetch_end` event that transitions it to the `ResWait` state, which is the home state for any server that is not actively a coordinator. The coordinator also locks the volume, after transitioning to the `Csub1-LockVol` state. When it is finished, it generates an `rs_lockandfetch_end` event that transitions it back to the `ResCoord` state.

If a directory is to be resolved, the coordinator transitions to the `DirRes` state after the volume is locked. While there, it coordinates a series of protocol phases described in Section 4.1.2. The phases are represented by the {`Csub,Sub`}`2-FetchLogs`, {`Csub,Sub`}`3-ParseLogs`, {`Csub,Sub`}`3.5-HandleInc`, {`Csub,Sub`}`4-InstallVV`, and {`Csub,Sub`}`5-MarkInc` states, respectively. States that begin with `Csub` are for the coordinator, and states that begin with `Sub` are for the subordinates. When the directory resolution is complete, the coordinator returns to the `ResCoord` state and instructs the DRG to unlock the volume, represented by the `Csub-UnlockVol` and `Sub-UnlockVol` states. On the other hand, if the ViceResolve request is for a file, the coordinator coordinates the volume-locking phase and transitions to the `FileRes` state, after which it conducts file resolution work. When the file resolution is complete, the Coordinator returns to the `ResCoord` state and instructs the DRG to unlock the volume. Upon completing either resolution request, the coordinator returns from the ViceResolve request and generates a `viceresolve_done` event that takes it to the `ResWait` state, where it is no longer a coordinator. After the ViceResolve request, the same process is repeated for each diverging object from the target volume, for which Venus requests resolution.

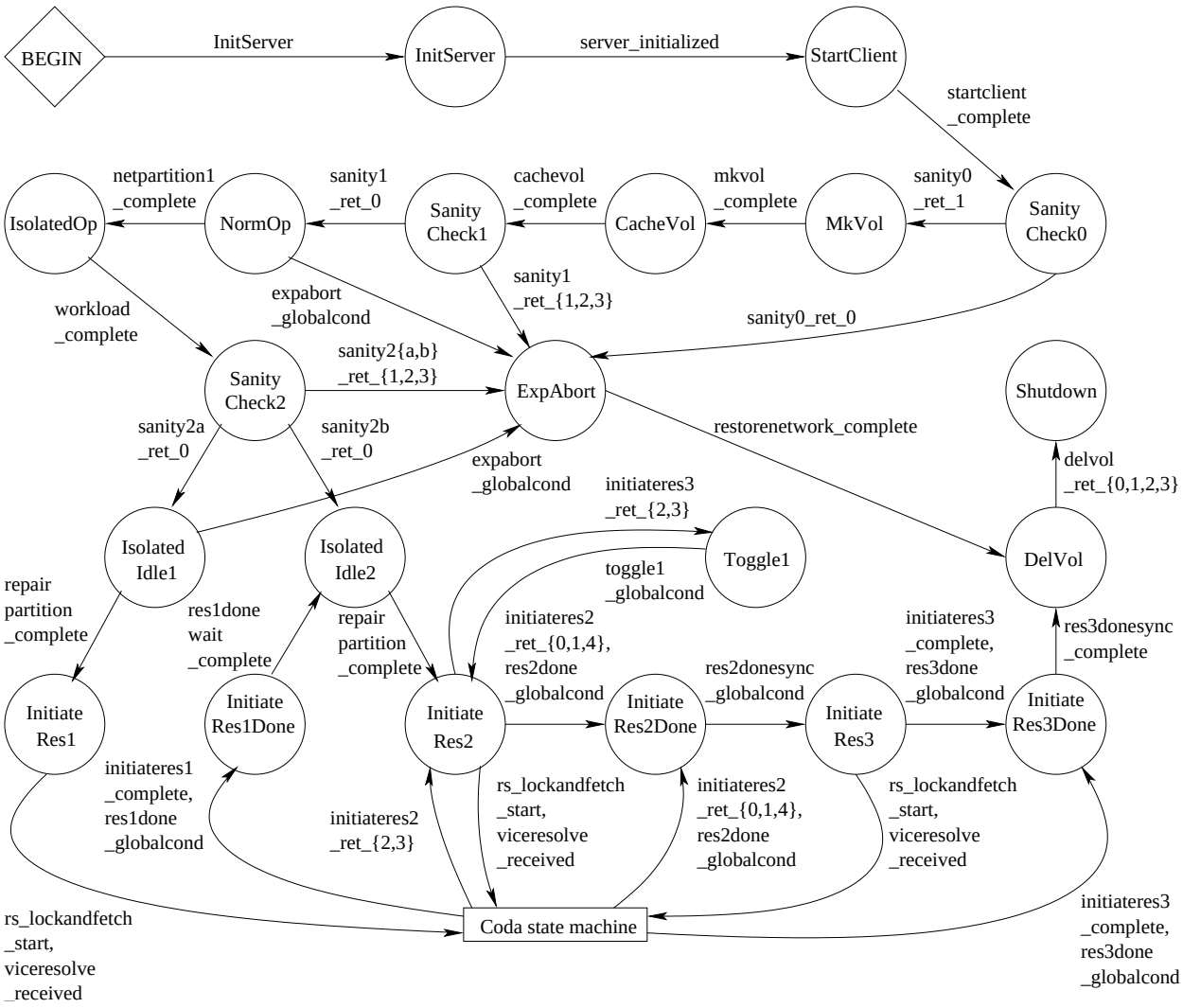


Figure 4.2 Supplemental State Machine

4.2.2.2 Supplemental state machine

The Coda state machine was augmented with a supplemental state machine that handles experiment control. The supplemental state machine is shown in Figure 4.2. The Coda state machine and the supplemental state machine are joined through the arcs in the figure that transition into and out of the box marked "Coda state machine." The supplemental state machine makes use of control modifiers, auxiliary states, and auxiliary events, as described in Section 3.1. In the description of the state machine below, unless otherwise stated, an action that is performed during a state is a direct result of a control modifier that was triggered by that state. Also, all of the states in the supplemental state machine, aside from the `InitServer` state, are auxiliary states. That makes sense, since the point of the supplemental state machine is to facilitate experiment control. Auxiliary events are

recognized by a _complete, _ret_x, or _globalcond at the end of their names.

Most of the supplemental state machine consists of a linear progression of steps through the experiment. It starts with the InitServer state, in which the state machine waits for the Coda server to initialize. Once all Coda servers have initialized, clients are started at each host during StartClient from a control modifier. SanityCheck0 is used to trigger a control modifier to make sure that the servers and clients have come up properly on each host. The sanity check generates a returnval event so the server will transition to the ExpAbort state and abort the experiment if the sanity check has failed. Once all of the servers have made it to the MkVol state, one of the servers uses a control modifier to create a replicated volume and populate it with data for the experiment. Then, in the CacheVol state, a control modifier makes sure that every client has a current view of the volume. Once the SanityCheck1 state has been reached, another sanity check is used to ensure that the volume has been properly created and that each client can properly access the volume. Assuming that the sanity check is passed, all of the servers meet in the NormOp state. The first network partition occurs while servers are in NormOp. The time when the partition took place can be determined because a netpartition1_complete auxiliary event would have been generated when the partition completed, and all of the state machines would have progressed to the IsolatedOp state. As mentioned in Section 4.2.1, a workload is applied to the volume during the first network partition, i.e., during the IsolatedOp state. Upon completion of its workload, a server transitions to the SanityCheck2 state, in which a control modifier confirms that the workload has been performed correctly. If the setup is not completely correct at the SanityCheck2 state, then the experiment is again aborted. Following a successful sanity check, the state machine transitions to IsolatedIdle1 if the experiment requires a correlated network fault, and IsolatedIdle2 in the baseline case. From both of those states, the network partition is repaired. Experiments from study B then transition to the InitiateRes2 state, while experiments from P3 and P4 transition to the InitiateRes1 state.

From either state, an initiateres$x$ control modifier instructs one of the clients to request that the replicated volume be resolved. The request occurs when all of the file servers are in an InitiateRes$x$ state. The client's request sends a ViceResolve call to one server, which generates a viceresolve_received event for the server. The event transitions the server to the Coda state machine to become a coordinator. When the other servers receive the coordinator's request to lock the volume during the locking phase for the ViceResolve call, they generate an rs_lockandfetch_start event and transition to the Coda state machine as subordinates. The resolution process is completed or aborted with an initiateres$x$ auxiliary event that causes the server to return to the respective InitiateRes$x$Done state. For studies

P3 and P4, correlated network partitions are injected from within the resolution initiated by `InitiateRes1` as described in the following subsection. After the `InitiateRes1Done` state, all of the file servers synchronize in the `InitiateRes2` state. In all studies, `InitiateRes2` is a state in which resolution is started (by the same client as for `InitiateRes1`, if applicable) and allowed to run to completion. Upon completion of the initiateres2 control modifier, all of the file servers synchronize in the `InitiateRes3` state, in which a final resolution request is used to update all clients with the contents of the resolved volume. `DelVol` uses a control modifier to clean up the volume, and during `Shutdown`, the experiment is completed, and servers and clients are shut down with a control modifier.

4.2.3 Faults and control modifiers

This subsection presents the faults and control modifiers that are used in the Coda case study. Tables 4.1 and 4.2 present the global state triggers that are used. Each entry in those tables specifies the name of a fault (or control modifier), the host(s) at which it is injected, its trigger condition, and whether it is to be injected one time ("once") or every time ("always") that the fault trigger transitions from false to true. Tables 4.3 and 4.4 summarize what all of the faults (and control modifiers) do. Tables 4.3 and 4.4 each have three columns. The first column gives the name of a fault (or control modifier). The second column specifies two parameters for the fault. The first parameter is a choice between 'e' and 'i,' and determines if the fault is an external or internal fault. The second parameter is a choice between 'n,' 'c,' 'r,' and 'g.' It determines the type of auxiliary event that the fault generates. 'n' designates no auxiliary event; 'c' designates a complete event; 'r' designates a returnval event; and 'g' designates a globally conditioned event. The last column is a description of what the associated fault (or control modifier) does.

The case study's two faults, netpartition1 and netpartition2, are presented at the tops of Tables 4.1 and 4.3. Both faults make use of the network fault injector, described in Section 3.5, to create network partitions. The network partitions are designed to block only Coda traffic, as described in Table 4.3. netpartition2 is omitted from study B, as study B does not contain a correlated network partition. For studies P3 and P4, x and *phasename* are replaced with appropriate values in the fault trigger for netpartition2. For P3 the values are `3` and `ParseLogs`, and for P4 they are `4` and `InstallVV`. The netpartition2 trigger causes the correlated network partition to be injected during Phase 3 of Coda's directory resolution protocol for P3, and during Phase 4 for P4.

The four tables also contain control modifiers. In particular, sanity checks are performed with the sanity0, sanity1, sanity2a, and sanity2b control modifiers. Studies P3 and P4 use

Table 4.1 Fault and Control Modifier Triggers

Name	Host	Trigger	once/ always
Faults			
netpartition1	h1,h2	((h1-server:NormOp)&(h2-server:NormOp))	once
netpartition2	h1	(((h1-server:Csub$x$-$phasename$)& (h2-server:Sub$x$-$phasename$))| ((h2-server:Csub$x$-$phasename$)& (h1-server:Sub$x$-$phasename$)))	once
Control Modifiers			
startclient	h1,h2	((h1-server:StartClient)& (h2-server:StartClient))	once
sanity0	h1,h2	((h1-server:SanityCheck0)& (h2-server:SanityCheck0))	once
mkvol	h1,h2	((h1-server:MkVol)&(h2-server:MkVol))	once
cachevol	h1,h2	((h1-server:CacheVol)&(h2-server:CacheVol))	once
sanity1	h1,h2	((h1-server:SanityCheck1)& (h2-server:SanityCheck1))	always
workload	h1,h2	((h1-server:IsolatedOp)& (h2-server:IsolatedOp))	once

sanity2a, but not sanity2b. On the other hand, study B uses sanity2b, but not sanity2a, so that study B can skip the group of `InitiateRes1` states and move right to the `InitiateRes2` state. One of the most important control modifiers is initiateres2. It requests resolution of the entire replicated volume after all of the true faults have been injected. Most of the measurements for the Coda case study were taken from the resolution that initiateres2 requested. initiateres2 is triggered when `h1-server` is in the `InitiateRes2` state, and `h2-server` resides in either the `InitiateRes2` or `ResWait` state. initiateres2 can be injected multiple times in a row if its return value indicates an intermediate resolution result. When initiateres2 finishes, it generates a returnval event indicating the status of the volume resolution. The return value indicates the status of the resolution. Once resolution has completed, an `initiateres2_ret_`x event brings the server at host h1 out of the Coda state machine and back to the supplemental state machine. Since the initiateres2 control modifier is injected only at host h1, the auxiliary event is generated only there. Thus, the res2done control modifier is used at host h2 to generate a globally conditioned event, `res2done_globalcond`. This event is triggered after the server on host h1 is done injecting initiateres2 control modifiers, and brings the server at host h2 out of the Coda state machine, to synchronize with host h1 in the supplemental state machine. initiateres1 is injected at host h1 and initiateres3 is

Table 4.2 More Control Modifier Triggers

Name	Host	Trigger	once/ always
Control Modifiers			
sanity{2a,2b}	h1,h2	((h1-server:SanityCheck2)& (h2-server:SanityCheck2))	always
repairpartition	h1,h2	(((h1-server:IsolatedIdle1)& (h2-server:IsolatedIdle1))| ((h1-server:IsolatedIdle2)& (h2-server:IsolatedIdle2)))	always
initiateres1	h1	((h1-server:InitiateRes1)& (h2-server:InitiateRes1))	once
res1done	h2	((h1-server:InitiateRes1Done)& (h2-server:ResWait))	once
res1donewait	h1,h2	((h1-server:InitiateRes1Done)& (h2-server:InitiateRes1Done))	once
initiateres2	h1	(((h1-server:InitiateRes2)& (h2-server:InitiateRes2))| ((h1-server:InitiateRes2)& (h2-server:ResWait)))	always
toggle1	h1	((h1-server:Toggle1))	always
res2done	h2	((h1-server:InitiateRes2Done)& ((h2-server:ResWait)| (h2-server:InitiateRes2)))	once
res2donesync	h1,h2	((h1-server:InitiateRes2Done)& (h2-server:InitiateRes2Done))	once
initiateres3	h2	((h1-server:InitiateRes3)& (h2-server:InitiateRes3))	once
res3done	h1	((h2-server:InitiateRes3Done)& ((h1-server:ResWait)| (h1-server:InitiateRes3)))	once
res3donesync	h1,h2	((h1-server:InitiateRes3Done)& (h2-server:InitiateRes3Done))	once
delvol	h1,h2	((h1-server:DelVol)&(h2-server:DelVol))	once
shutdown	h1,h2	((h1-server:Shutdown)&(h2-server:Shutdown))	once
expabort	h1 h2	((h2-server:ExpAbort)&((h1-server:NormOp)| (h1-server:IsolatedIdle1))) ((h1-server:ExpAbort)&((h2-server:NormOp)| (h2-server:IsolatedIdle1)))	always
restorenetwork	h1,h2	((h1-server:ExpAbort)&(h2-server:ExpAbort))	always

Table 4.3 Fault and Control Modifier Descriptions

Name	{e|i}, {n|c|r|g}	Description
Faults		
netpartition1	i,c	The network fault injector is used to create a network partition. Host h1 blocks all traffic coming from host h2 using TCP or UDP on ports 2430 to 2433, i.e., Coda network traffic. Host h2 does the same but blocks traffic from host h1.
netpartition2	i,n	The network fault injector is used to create a network partition. Host h1 blocks all traffic coming from or going to host h2 using TCP or UDP on ports 2430 to 2433, i.e., Coda network traffic.
Control Modifiers		
startclient	e,c	At each host, a Venus client is started and authentication tokens are acquired.
sanity0	e,r	The client at each host checks to make sure that the target volume does not already exist.
mkvol	e,c	The client at host h1 creates, mounts, and populates the target volume. Nothing is done at host h2.
cachevol	e,c	Both clients update their AVSGs and cache the target volume with an `ls -R` command.
sanity1	e,r	Both clients check if the target volume has been set up correctly.
workload	e,c	Each client applies an independent workload to the target volume, whose differences are automatically resolvable.
sanity{2a,2b}	e,r	Each client checks that the local replica of the target volume matches an appropriate golden directory, depending on the work performed by the workload control modifier.
repairpartition	i,c	The network fault injector is used at both hosts to flush all IP Chains rules.
initiateres1	e,c	The client at h1 updates its AVSG and uses `ls -R` to initiate resolution of the target volume.
res1done	e,g	No action is taken.
res1donewait	e,c	No action is taken.
initiateres2	e,r	The client at h1 updates its AVSG and reads the contents of the target volume to initiate resolution. It also compares the resolution result to a golden directory.
toggle1	e,g	No action is taken.
res2done	e,g	No action is taken.
res2donesync	e,g	No action is taken.

Table 4.4 More Control Modifier Descriptions

Name	{e|i}, {n|c|r|g}	Description
Control Modifiers		
initiateres3		The client at h2 updates its AVSG and uses `ls -R` to initiate resolution of the target volume.
res3done	e,g	No action is taken.
res3donesync	e,g	No action is taken.
delvol	e,r	The client at h1 deletes the contents of the target volume, unmounts it, and then removes it. Nothing is done at host h2.
shutdown	e,n	The clients and file servers are properly shut down at both hosts. Also, the Coda root volume is unmounted.
expabort	e,g	No action is taken.
restorenetwork	i,c	The network fault injector is used at both hosts to flush all IP Chains rules.

injected at host h2. Both control modifiers conclude with complete events that bring their respective hosts back to the supplemental state machine. Globally conditioned events are used for those two control modifiers to bring the noninjecting host out of the Coda state machine, as was the case with initiateres2. Control modifiers that do not perform actions are used for synchronization purposes. The remaining control modifiers are explained with Tables 4.1, 4.2, 4.3, and 4.4.

4.2.4 Coda instrumentation

Coda was instrumented for use with Loki, following the guidelines set forth in Section 3.4. First, `libappproxy.a` was linked into the Coda file server executable. In order to initialize the file server for use with Loki, we inserted the `INITIALIZE_PROXY_COMM` C macro at the beginning of the file server's `main()` routine. Then the `lokiNotifyEvent()` method was invoked throughout the file server source code in order to identify events that are used by Loki state machines. Phases of the directory resolution protocol were identified by methods responding to Coda RPC2 requests to perform work in the protocol. Finally, fault definition files were created for the faults and control modifiers described in the previous subsection.

CHAPTER 5

RESULTS

This chapter presents the results of the fault injection study. It begins with a discussion of the experimental setup and workload, followed by a discussion of proper experiments. Following those discussions, the remainder of the chapter describes measure definitions and experimental results.

5.1 Experimental Setup and Workload

After defining the fault injection study, we performed 200 experiments for each of the three studies: B, P3, and P4. Experiments were executed on two Pentium II, 233 MHz machines with 128 MB of RAM running the Linux 2.4.17 kernel. The machines were located on the same LAN using 100 Mb Ethernet. The following Coda packages were used: Coda 5.3.13, Coda kernel module 5.2.3, LWP 1.7, RPC2 1.11, and RVM 1.4. The target volume used in each experiment was populated with 16-100 B files, 10-1 KB files, 10-100 KB files, and 4-1 MB files. The workload performed during each partitioned update consisted of a series of the following types of actions: directory reads, file reads, file creations, file deletions, global replacements within a file, writes to the end of a file, and archives[1] of the volume. Again, these workloads were designed such that the partitioned updates could automatically be resolved correctly. When the experiments finished executing, they were passed through Loki's analysis process to create a global timeline for each experiment and to determine the correctness of fault injections.

[1]The archive was created with the Linux/Unix `tar` command.

5.2 Proper Experiment Filtering

The first detail to consider after performing experiments is the subset of experiments upon which measures will be computed. Experiments that do not set up correctly and ones that do not correctly inject faults should be filtered out. The supplemental state machine, presented in Section 4.2.2.2, is designed so that experiments that fail a sanity check (either sanity0, sanity1, sanity2a, or sanity2b) are aborted. Failure of any of the sanity checks at any state machine indicates that an experiment did not set up correctly and can be filtered out through a check of the returnval events produced by each of the sanity checks. In addition to proper experiment setup, the injection status of faults is also checked. Typically, Loki checks the injection status of all faults performed in an experiment. However, in the case study presented in this thesis, only the status of the netpartition2 fault needed to be checked. With the exception of the netpartition2 fault, when a state machine is in a state in which a fault or control modifier is to be injected, the state machine will wait in that state until the fault is injected. Therefore, if faults and control modifiers are injected, they will be injected correctly; no further analysis is required. Furthermore, if the sanity checks are passed and the netpartition2 fault is injected correctly, the experiment has progressed to the point that useful data can be gathered.

We perform proper experiment filtering with Loki's measures language by prepending each measure definition with the triples in Table 5.1. h1_sanitychk_tpl and h2_sanitychk_tpl check if the file server on hosts h1 and h2, respectively, ever failed a sanity check during an experiment. netpartition2_chk_tpl examines the fault injection status of the netpartition2 fault. Finally, filter_badexp_tpl uses the other three triples to filter out improper experiments. In the definitions of measures, proper experiment filtering is not mentioned; however, the triples from Table 5.1 are implicitly prepended to each definition.

5.3 Measure Definitions

There are three main groups of measures of interest, as described in the subsequent subsections. Each measure considered is applied to a single study; thus, a simple sampling campaign-level measure is implicitly defined for each study-level measure defined below.

5.3.1 Final resolution outcomes

The first group classifies the outcome of the complete resolution of the target volume following the final network partition for an experiment. In the supplemental state machine,

Table 5.1 Proper Experiment Filter

h1_sanitychk_tpl	
Subset	`(1)`
Predicate	`(h1-server,sanity0_ret_1)||(h1-server,sanity1_ret_0)|| (h1-server,sanity2a_ret_0)||(h1-server,sanity2b_ret_0)`
Observation Function	`if(count(UP,IMPULSE,EXP_START_TIME,EXP_END_TIME) == 3) {return 1.0;} else{return 0.0;}`

h2_sanitychk_tpl	
Subset	`(1)`
Predicate	`(h2-server,sanity0_ret_1)||(h2-server,sanity1_ret_0)|| (h2-server,sanity2a_ret_0)||(h2-server,sanity2b_ret_0)`
Observation Function	`if(count(UP,IMPULSE,EXP_START_TIME,EXP_END_TIME) == 3) {return 1.0;} else{return 0.0;}`

netpartition2_chk_tpl	
Subset	`(1)`
Predicate	`(1)`
Observation Function	`if(faultcheck(h1-server,netpartition2,CORRECT)){return 1.0;} else{return 0.0;}`

filter_badexp_tpl	
Subset	`(h1_sanitychk_tpl == 1.0 && h2_sanitychk_tpl == 1.0 && netpartition2_chk_tpl == 1.0)`
Predicate	`(1)`
Observation Function	`return 0.0;`

that resolution is initiated from the `InitiateRes2` state by the initiateres2 control modifier. In the baseline case, the resolution follows one network partition that sets up the resolution scenario. In all other cases (i.e., P3 and P4), it follows two network partitions, one that sets up the resolution scenario and one that interrupts the first resolution attempt.

There are three possible final outcomes for a complete volume resolution that is allowed to complete (i.e., one that is not interrupted by a network partition). First, the target volume can be automatically resolved correctly. In that scenario, Coda is able to merge the diverging replicas in the experiment automatically in a way that is consistent with partitioned updates. Since the workload of our experiment results in partitioned updates that are serializable, automatic and correct resolution is always possible. A second possible final outcome is that Coda will automatically resolve the target volume in an incorrect manner. In that situation, Coda will again be able to merge the diverging replicas automatically; however, the result will be a directory that is not consistent with the partitioned updates. The last possible

Table 5.2 Resolution Outcome Triple

res2chk_tpl	
Subset	`(1)`
Predicate	`(h1-server,initiateres2_ret_`*outcome*`)`
Observation Function	`if(count(UP,IMPULSE,EXP_START_TIME,EXP_END_TIME) == 1) {return 1.0;} else{return 0.0;}`

final outcome is that Coda will not be able to resolve the target volume automatically, thus producing a result requiring manual resolution.

For each of these resolution outcomes, it is possible to define a measure that outputs a 1 if the outcome occurs and 0 otherwise. The mean of this measure tells us the fraction of experiments that produce the desired outcome. The final resolution outcome can be identified by the returnval event that is generated by the initiateres2 control modifier. The measures for resolution outcomes are named res2_correct, res2_incorrect, and res2_manual, respectively. Each measure is constructed from the res2chk_tpl triple in Table 5.2 through replacement of *outcome* with the appropriate return value indicating the desired resolution outcome.

5.3.2 Temporarily unavailable resources

In addition to the above three possible final resolution outcomes, it could happen that a temporary unavailability of resources prevents Coda from producing a final resolution result. When that occurs, an intermediate result noting the unavailability of resources is generated. The result is only an intermediate result because if the client continues to request that the target volume be resolved, one of the aforementioned final resolution results finally happens. The second group of measures deals with the possibility of temporarily unavailable resources.

Five measures concerning unavailable resources are computed: res2_unavailable, res2_count, res2_count_g2, correct_if_g1_res2, and g1_res2_if_correct. res2_unavailable measures whether an experiment experiences temporarily unavailable resources when the initiateres2 control modifier is invoked, with a value of 1 if the experiment experiences unavailable resources, and 0 otherwise. We construct its definition from the res2chk_tpl triple in Table 5.2 by replacing *outcome* with the return value indicating temporarily unavailable resources. The mean of this measure is the fraction of experiments that see unavailable resources. The second measure, res2_count, is the number of times that initiateres2 must be invoked by the client before producing a final resolution result. The definition of res2_count is described by the res2_cnt_tpl triple in Table 5.3. The predicate in that triple uses an

Table 5.3 Multiple Resolution Triples

res2_cnt_tpl	
Subset	`(1)`
Predicate	`forany({initiateres2_ret_0,initiateres2_ret_1, initiateres2_ret_2,initiateres2_ret_3,initiateres2_ret_4}, [x,(h1-server,x)])`
Observation Function	`return count(UP,IMPULSE,EXP_START_TIME,EXP_END_TIME);`

res2_g2_tpl	
Subset	`(1)`
Predicate	`(1)`
Observation Function	`if(res2_cnt_tpl > 2){return 1.0;} else{return 0.0;}`

correct_if_g1_tpl	
Subset	`(res2_cnt_tpl > 1)`
Predicate	`(1)`
Observation Function	`return res2chk_tpl;`

g1_if_correct_tpl	
Subset	`(res2chk_tpl == 1.0)`
Predicate	`(1)`
Observation Function	`if(res2_cnt_tpl > 1.0){return 1.0;} else{return 0.0;}`

existential quantifier over all possible resolution outcomes (temporary and final) to produce impulses in the predicate timeline whenever an initiateres2 control modifier completes. The impulses are counted to determine the number of initiateres2 control modifiers that took place. The third measure returns 1 if the res2_count is greater than 2, and 0 otherwise. It uses the res2_cnt_tpl triple from Table 5.3 to count the number of initiateres2 calls and then res2_g2_tpl to see if the count is greater than 2.

The last two measures in the group, identified by temporarily unavailable resources, are similar. Each of these last two measures filters experiments beyond the proper experiment filter. correct_if_g1_res2 filters out all experiments that do not observe more than one initiateres2 call. The remaining experiments return 1 if initiateres2 produces an automatic and correct resolution result, and 0 otherwise. On the other hand, g1_res2_if_correct filters out all experiments whose initiateres2 calls do not result in automatic and correct resolution. For the remaining experiments, the measure is 1 if the experiment observes more than 1 initiateres2, and 0 otherwise. The means of the last two measures give the fraction of unfiltered

experiments that meet the defined condition. Their definitions make use of the res2chk_tpl triple from Table 5.2 and the res2_cnt_tpl triple from Table 5.3. Those triples are appended with the correct_if_g1_tpl triple for the correct_if_g1_res2 measure and the g1_if_correct_tpl triple for the g1_res2_if_correct measure.

5.3.3 End-to-end measures

The third group of measures pertains to end-to-end measures from the client's point of view. The end-to-end result is measured when the target volume is correctly resolved automatically (end2end_correct) and when its result requires manual resolution (end2end_manual). Each of the measures is a sum of intervals pertaining to the time the client spends waiting for the target volume to be resolved during each initiateres2. Each of these intervals is measured from the time that one of the Coda servers receives the first ViceResolve request until initiateres2 returns with an exit value (i.e., produces a returnval event). Measuring is not started until the first ViceResolve request has been received, because at the beginning of initiateres2 the client checks which servers are available. The check is not part of the resolution process, and takes a nontrivial amount of time.

end2end_correct and end2end_manual are defined using the res2chk_tpl triple from Table 5.2 appended with end2end_tpl from Table 5.4. *outcome* is replaced in both triples with the return value of the desired resolution outcome. The end2end_tpl triple computes an initial estimate of the measure by summing the length of a set of intervals. Each interval starts when the file server on host h1 leaves the `Initiateres2` state and ends with the associated returnval event generated by the initiateres2 control modifier injected during the `Initiateres2` state. This initial estimate would be correct if the file server were the coordinator for the first resolution request generated by each initiateres2 control modifier. The file server on h1 would receive the first ViceResolve request generated as a result of an initiateres2 control modifier, and that ViceResolve request would generate the event that transitioned the file server on host h1 out of the `InitiateRes2` state. However, if the file server on h2 is the coordinator for the first resolution request generated by an initiateres2 control modifier, then the ViceResolve will occur before the file server on host h1 leaves the `Initiateres2` state. It will leave the `Initiateres2` state when it generates a `rs_lockandfetch_start` event due to the file server on host h2 requesting that it lock the target volume. The second part of the end2end_tpl observation function accounts for the case where `h2-server` receives the first ViceResolve request generated from an initiateres2 control modifier.

In addition, the same end-to-end measures are computed minus the time at the beginning of the measure spent waiting for a volume lock to be released from a previously failed resolu-

Table 5.4 Base End-to-End Triple

end2end_tpl	
Subset	(res2chk_tpl == 1.0)
Predicate	(h1-server,InitiateRes2)||(h1-server,initiateres2_ret_*outcome*) ||((h2-server,viceresolve_received)&&(h1-server,InitiateRes2))

Observation Function	/* Compute initial estimate. */
	double last_ret =
	  instant(UP,IMPULSE,1,EXP_START_TIME,EXP_END_TIME);
	double first_ir2_state =
	  instant(UP,STEP,1,EXP_START_TIME,EXP_END_TIME);
	double ir2_state_time =
	  total_duration(TRUE,EXP_START_TIME,EXP_END_TIME);
	double mymeasure =
	  (last_ret - first_ir2_state) - ir2_state_time;
	/* If h2 is the coordinator, add in initial part. */
	int dimpulse_cnt =
	  count(DOWN,IMPULSE,EXP_START_TIME,EXP_END_TIME);
	int dstep_cnt = count(DOWN,STEP,EXP_START_TIME,EXP_END_TIME);
	double dimpulse_time, ustep_time, dstep_time;
	int jjj = 1;
	for (int iii=1; iii<=dstep_cnt; iii++){
	  ustep_time =
	  instant(UP,STEP,iii,EXP_START_TIME,EXP_END_TIME);
	  dstep_time =
	  instant(DOWN,STEP,iii,EXP_START_TIME,EXP_END_TIME);
	  while ((jjj <= dimpulse_cnt)&&((dimpulse_time =
	  instant(DOWN,IMPULSE,jjj,EXP_START_TIME,EXP_END_TIME))
	  < ustep_time))
	  {jjj++;}
	  if (jjj > dimpulse_cnt){break;}
	  else if (ustep_time < dimpulse_time &&
	  dimpulse_time < dstep_time)
	  {mymeasure = mymeasure + (dstep_time - dimpulse_time);}
	}
	return mymeasure;

tion attempt. To perform this computation, the start of the first initiateres2 is found. Then, for each server, the first point after the first initiateres2 when the server is able to lock the volume is determined. If a server has yet to release a lock when the first initiateres2 is called, several `rs_lockandfetch_error` events are observed before the first `rs_lockandfetch_end` event. The `rs_lockandfetch_end` event is the point at which the server locks the volume successfully. After the first point at which both servers are able to successfully lock the volume is determined, the measure backtracks in the global timeline to find the immediately previous ViceResolve request that arrived at any server; starting at that point, measuring of the end-to-end measure is performed just as before. The new end-to-end measure is made for automatic and correct resolutions (end2end-lockerr_correct) and for manual resolutions (end2end-lockerr_manual).

The definitions for those two measures make use of the triples in Table 5.5 to determine when the following occur: the first initiateres2 control modifier, the first successful lock by the file server on host h1, the first successful lock by the file server on host h2, the first point at which both file servers have successfully locked the target volume, and the occurrence of the ViceResolve call that results in the first successful lock of the target volume by both servers. The occurrence of the ViceResolve call that results in the first successful lock of the target volume by both servers is then used by end2end_nolockerr_tpl from Table 5.6 as the starting point of the end-to-end measure. As with end2end_tpl from Table 5.4, *outcome* is replaced with the return value that is appropriate for the resolution outcome pertaining to the measure. The execution of end2end_nolockerr_tpl is very similar to that of end2end_tpl.

Finally, for experiments that observe multiple initiateres2 calls, end2end_correct and end2end_manual can be subdivided into the parts attributed to each initiateres2 call. The subdivided measures will be called end2end_correct_x and end2end_manual_x, where x indicates that the measure pertains to the x^{th} initiateres2 call. end2end_correct_1 and end2end_manual_1 can be computed using the res2chk_tpl triple from Table 5.2 appended with the end2end_1_tpl triple from Table 5.7, and using the appropriate return value for the desired resolution outcome. end2end_1_tpl is constructed much like end2end_tpl from Table 5.4 except that it limits the measure to the resolution attributed to the first initiateres2 control modifier. Likewise, end2end_correct_2 and end2end_manual_2 are constructed using the res2chk_tpl triple from Table 5.2 appended with the end2end_2_tpl triple from Table 5.8, using the appropriate return value for the desired resolution outcome.

Table 5.5 Helper Lock Error Triples

first_initiateres2_tpl	
Subset	`(res2chk_tpl == 1.0)`
Predicate	`(h1-server,initiateres2)`
Observation Function	`return instant(UP,IMPULSE,1,EXP_START_TIME,EXP_END_TIME);`

first_successful_lock_h1_tpl	
Subset	`(1)`
Predicate	`(h1-server,rs_lockandfetch_end,first_initiateres2_tpl, EXP_END_TIME)`
Observation Function	`return instant(UP,IMPULSE,1,EXP_START_TIME,EXP_END_TIME);`

first_successful_lock_h2_tpl	
Subset	`(1)`
Predicate	`(h2-server,rs_lockandfetch_end,first_initiateres2_tpl, EXP_END_TIME)`
Observation Function	`return instant(UP,IMPULSE,1,EXP_START_TIME,EXP_END_TIME);`

first_successful_lock_tpl	
Subset	`(1)`
Predicate	`(1)`
Observation Function	`if (first_successful_lock_h1_tpl > first_successful_lock_h2_tpl) {return first_successful_lock_h1_tpl;} else{return first_successful_lock_h2_tpl;}`

measure_start_tpl	
Subset	`(1)`
Predicate	`({h1-server,h2-server},[x,(x,viceresolve_received, first_initiateres2_tpl,first_successful_lock_tpl)])`
Observation Function	`int last_viceresolve = count(UP,IMPULSE,EXP_START_TIME, EXP_END_TIME); return instant(UP,IMPULSE,last_viceresolve,EXP_START_TIME, EXP_END_TIME);`

Table 5.6 End-to-End Triple Without Locking Errors

end2end_nolockerr_tpl	
Subset	(res2chk_tpl == 1.0)
Predicate	(h1-server,InitiateRes2,first_successful_lock_tpl, EXP_END_TIME)||(h1-server,initiateres2_ret_*outcome*)|| ((h2-server,viceresolve_received,first_successful_locktpl, EXP_END_TIME)&&(h1-server,InitiateRes2, first_successful_lock_tpl,EXP_END_TIME))

Observation Function	/* Compute initial estimate. */
	double last_ret =
	  instant(UP,IMPULSE,1,EXP_START_TIME,EXP_END_TIME);
	double ir2_state_time =
	  total_duration(TRUE,EXP_START_TIME,EXP_END_TIME);
	double mymeasure =
	  (last_ret - measure_start_tpl) - ir2_state_time;
	/* If h2 is the coordinator, add in initial part. */
	int dimpulse_cnt =
	  count(DOWN,IMPULSE,EXP_START_TIME,EXP_END_TIME);
	int dstep_cnt = count(DOWN,STEP,EXP_START_TIME,EXP_END_TIME);
	double dimpulse_time, ustep_time, dstep_time;
	int jjj = 1;
	for (int iii=1; iii<=dstep_cnt; iii++){
	  ustep_time =
	  instant(UP,STEP,iii,EXP_START_TIME,EXP_END_TIME);
	  dstep_time =
	  instant(DOWN,STEP,iii,EXP_START_TIME,EXP_END_TIME);
	  while ((jjj <= dimpulse_cnt)&&((dimpulse_time =
	  instant(DOWN,IMPULSE,jjj,EXP_START_TIME,EXP_END_TIME))
	  < ustep_time))
	  {jjj++;}
	  if (jjj > dimpulse_cnt){break;}
	  else if (ustep_time < dimpulse_time &&
	  dimpulse_time < dstep_time)
	  {mymeasure = mymeasure + (dstep_time - dimpulse_time);}
	}
	return mymeasure;

Table 5.7 Triple for End-to-End Due to First initiateres2

end2end_1_tpl	
Subset	`(res2chk_tpl == 1.0)`
Predicate	`(h1-server,InitiateRes2)||(h1-server,initiateres2_ret_`*outcome*`)` `||((h2-server,viceresolve_received)&&(h1-server,InitiateRes2))`

Observation Function	`/* Compute initial estimate. */`
	`double first_dstep =`
	`  instant(DOWN,STEP,1,EXP_START_TIME,EXP_END_TIME);`
	`double second_ustep =`
	`  instant(UP,STEP,2,EXP_START_TIME,EXP_END_TIME);`
	`double mymeasure =`
	`  second_ustep - first_dstep;`
	`/* If h2 is the coordinator, add in initial part. */`
	`int dimpulse_cnt =`
	`  count(DOWN,IMPULSE,EXP_START_TIME,EXP_END_TIME);`
	`double first_ustep =`
	`  instant(UP,STEP,1,EXP_START_TIME,EXP_END_TIME);`
	`double dimpulse =`
	`  instant(DOWN,IMPULSE,1,EXP_START_TIME,EXP_END_TIME);`
	`int jjj = 1;`
	`while ((jjj <= dimpulse_cnt)&&((dimpulse =`
	`  instant(DOWN,IMPULSE,jjj,EXP_START_TIME, EXP_END_TIME))`
	`< first_ustep))`
	`{jjj++;}`
	`if (jjj > dimpulse_cnt){}`
	`else if (first_ustep < dimpulse && dimpulse < first_dstep)`
	`{mymeasure = mymeasure + (first_dstep - dimpulse);}`
	`return mymeasure;`

Table 5.8 Triple for End-to-End Due to Second initiateres2

end2end_2_tpl	
Subset	(res2chk_tpl == 1.0)
Predicate	(h1-server,InitiateRes2)||(h1-server,initiateres2_ret_*outcome*) ||((h2-server,viceresolve_received)&&(h1-server,InitiateRes2))

Observation Function	/* Compute initial estimate. */
	double second_dstep =
	instant(DOWN,STEP,2,EXP_START_TIME,EXP_END_TIME);
	double first_uimpulse
	instant(UP,IMPULSE,1,EXP_START_TIME,EXP_END_TIME);
	double mymeasure =
	first_uimpulse - second_dstep;
	/* If h2 is the coordinator, add in initial part. */
	int dimpulse_cnt =
	count(DOWN,IMPULSE,EXP_START_TIME,EXP_END_TIME);
	double second_ustep =
	instant(UP,STEP,2,EXP_START_TIME,EXP_END_TIME);
	double dimpulse =
	instant(DOWN,IMPULSE,1,EXP_START_TIME,EXP_END_TIME);
	int jjj = 1;
	while ((jjj <= dimpulse_cnt)&&((dimpulse =
	instant(DOWN,IMPULSE,jjj,EXP_START_TIME, EXP_END_TIME))
	< second_ustep))
	{jjj++;}
	if (jjj > dimpulse_cnt){}
	else if (second_ustep < dimpulse && dimpulse < second_dstep)
	{mymeasure = mymeasure + (second_dstep - dimpulse);}
	return mymeasure;

Table 5.9 Resolution Outcome Results

Study	# of Experiments	res2_correct Mean	res2_incorrect Mean	res2_manual Mean
B	75	100.0%	0.0%	0.0%
P3	68	100.0%	0.0%	0.0%
P4	46	52.2%	0.0%	47.8%

5.4 Experimental Results

The experimental results pertaining to each of the measure groups defined in Section 5.3 are now presented.

5.4.1 Final resolution outcomes

First, the outcome of the complete resolution of the target volume and its contents following an experiment's final network partition is considered; that outcome results from the resolution requested by initiateres2. As mentioned in Section 5.3, three possible outcomes are considered: (1) automatic and correct resolution, (2) automatic and incorrect resolution, and (3) manual resolution. Table 5.9 shows our findings. One of the most notable results is that none of the studies had experiments resulting in incorrect resolutions. That is significant because it tells us that although some resolutions required manual resolution, none of the final results are incorrect from a data integrity point of view. Next, it is observed that the baseline case always resulted in automatic and correct resolution, which was expected. For study P3, the results show that all of the resolutions performed automatically and correctly. In the P3 experiments, the directory resolution protocol was aborted when the correlated network partition, which occurred in Phase 3 of the directory resolution protocol, was detected by the Coda servers. When the network was repaired and the client requested resolution again, during initiateres2, the directory resolution protocol was restarted. This behavior is as described in [6], and our experiments serve to verify it.

Lastly, the final resolution outcomes from study P4 are examined. In this study, a correlated fault was injected during Phase 4 of the directory resolution protocol, which is responsible for committing the directory resolution process and converging on a storeid for it. P4 is the only study in which resolution resulted in something other than automatic and correct resolution. In fact, 47.8% of the time, the outcome required manual resolution. That is a very significant result because of its potential impact on the volume's availability. Once Coda has declared that manual resolution is required, a human must step in to converge the

Table 5.10 Unavailable Resource Results

Measure	Study	# of Experiments	Mean	Std Dev
res2_unavailable	B	75	0.0%	na
res2_unavailable	P3	68	0.0%	na
res2_unavailable	P4	46	52.2%	na
res2_count	B	75	1.0	0.0
res2_count	P3	68	1.0	0.0
res2_count	P4	46	1.5	0.5
res2_count_g2	P4	46	0.0%	na
correct_if_g1_res2	P4	24	100.0%	na
g1_res2_if_correct	P4	24	100.0%	na

diverging replicas, and that could take a significant amount of time. Using Loki's measure support, it was confirmed that in each of the experiments from study P4 that require manual resolution, the interrupted Phase 4 resulted in an `rs_installvv_error` event for at least one of the servers. Then, when directory resolution was restarted, the protocol omitted Phase 4 and performed Phase 5, in which the volume was marked inconsistent, i.e., in need of manual resolution. On the other hand, in P4's experiments that resulted in automatic and correct resolution, none of the servers generated an `rs_installvv_error` event. In fact, during the first resolution attempt, the directory resolution protocol proceeded to the point that resolution of the directory was committed despite the first network partition. However, not all of the files in the target volume were resolved. Thus, when initiateres2 was performed for the second time, the remaining unresolved files were resolved.

5.4.2 Temporarily unavailable resources

Next, we consider the case in which resolution experiences temporarily unavailable resources. Associated results are listed in Table 5.10. First, it can be seen that only study P4 ever experienced unavailable resources. Thus, exactly one initiateres2 was invoked for studies B and P3. The temporarily unavailable resource condition happened no more than once, resulting in at most two initiateres2 calls. The final observation is that temporarily unavailable resources occurred in study P4 if and only if resolution started by initiateres2 resulted in automatic and correct resolution. We arrived at that conclusion because the means of both correct_if_g1_res2 and g1_res2_if_correct are 100.0%. Thus, experiments from study P4 either required manual resolution or experienced temporarily unavailable resources and then resulted in correct and automatic resolution.

Table 5.11 End-to-End Results

Study	*class*	# of Experiments	end2end_*class*		end2end-lockerr_*class*	
			Mean (s)	Std Dev (s)	Mean (s)	Std Dev (s)
B	correct	75	21.0	0.3	21.0	0.3
P3	correct	68	287.8	14.2	21.9	3.3
P4	correct	24	258.8	3.2	23.6	2.1
P4	correct_1	24	240.5	3.0	-	-
P4	correct_2	24	18.3	0.4	-	-
P4	manual	22	248.2	7.9	0.64	0.02

5.4.3 End-to-end results

The final group of measures considered are end-to-end measures from the client's point of view. The measures obtained are listed in Table 5.11 and graphically displayed in Figure 5.1. The figure presents end-to-end measures for four cases. Each case is defined by its study and its final resolution outcome, and is labeled appropriately across the bottom of the figure. For each case, the end-to-end measure is broken down into subcomponents in two ways. The first breakdown distinguishes which portions of the measure resulted from the first initiateres2 call versus a second initiateres2 call. As mentioned previously, a second initiateres2 call is only present in experiments from study P4 that resulted in automatic and correct resolution. The percentage noted above the top of the bar representing this breakdown is the percentage of the end-to-end measure that is due to a second initiateres2 call. The percentage noted below the bottom of the bar is the percentage due to the first initiateres2 call. The second breakdown is labeled like the first, but distinguishes between time spent waiting for the already locked volume to unlock versus time during which the system is successfully performing resolution work.

There are several points to consider. Most noticeably, studies containing correlated network partitions require significantly more time to perform resolution than studies without such partitions. The reason is that attempts to lock the target volume during the second resolution fail because the volume is still locked from the first resolution. In fact, the end-to-end measure for cases involving correlated network partitions is nearly 12 to 14 times larger than for the baseline case. The problem is that the subordinate, `h2-server`, maintains its lock on the target volume from the first resolution attempt, even though the protocol has been aborted due to the network partition. Using Loki measures support, we have confirmed that this phenomenon occurred in all the experiments in studies P3 and P4. The time spent waiting on the volume's lock causes the client to block, thus having a negative impact on

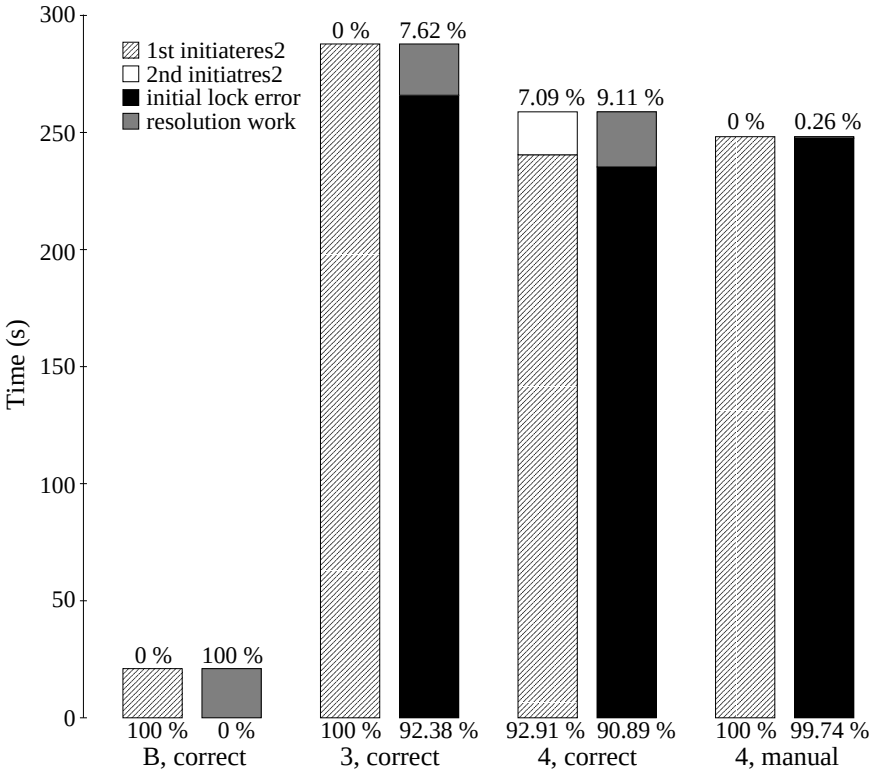


Figure 5.1 End-to-End Comparison

the availability of the volume. According to [6, p. 71], if the coordinator crashes[2] then the subordinate unlocks its volume replica. If it does not unlock the volume, then when resolution is restarted, the protocol will block in Phase 1, which is precisely the behavior that we observed. The subordinate continually probes the coordinator in order to detect crashes (or network partitions) [6, p. 71]. If the coordinator does not respond, the subordinate unlocks the volume; however, in our experiments, the subordinate did not unlock the volume until some time much later, presumably due to a timer expiration for the lock. When a subordinate is asked by a server to lock a volume, and the volume has already been locked because of a request from the same server, we hypothesize that the protocol could be improved if the subordinate released and relocked the volume. The subordinate could keep track of who asked it to lock the volume and for what purpose; if the same server later asked the subordinate to lock the volume, and the subordinate still had the volume locked from the first request, then the subordinate could unlock and relock the volume. That would allow the large blocking time that is observed in the end-to-end results to be avoided.

Another observation is that for experiments from study P4 that require manual resolution, the time spent performing resolution work is much smaller than in any other scenario. That

[2]As far as the subordinate is concerned, a coordinator crash is the same as a network partition.

is significant, because it means that if manual resolution is required, that fact is quickly detected and reported.

The final observation from the end-to-end results concerns experiments from study P4 that result in automatic and correct resolution. As previously mentioned, those experiments always experienced temporarily unavailable resources, thus requiring the client to request resolution twice. The interesting thing is that the end-to-end measure was shorter for P4's experiments that resulted in automatic and correct resolution than for P3's experiments that resulted in automatic and correct resolution, even though P4's experiments are the ones that experienced temporarily unavailable resources. While the cause of that phenomenon is unclear, it does provide some insight into the nature of the temporarily unavailable resources. It would appear that the unavailability of resources is not simply due to a timer, or it would be expected that the end-to-end measure would be larger for study P4. In addition, Figure 5.1 shows that in the correct case, some resolution work was being done during the first initiateres2 call. That may indicate that the correlated network partition caused some of the file system's resources to be temporarily tied up during the first initiateres2 call, requiring a second initiateres2.

CHAPTER 6

CONCLUSIONS

In this thesis, we detail how a system is evaluated using the Loki fault injector and present an experimental evaluation of the Coda distributed file system using Loki. The first half of the thesis focuses on Loki and begins with a review of the Loki fault injector. This review includes the design and operation of Loki's runtime and its language for specifying measures to be computed from a fault injection campaign. The review is followed by details concerning the use of Loki to evaluate a system. These details include the specification of a fault injection campaign, instrumentation of the system under study, and incorporation of fault types into a Loki fault library. Then, we present a particular approach to using Loki that involves a lightweight fault injection proxy. This approach was designed because of threading conflicts that arose when we were attempting to incorporate Loki's runtime directly into Coda's file servers. The proxy design is general so that it can be used to circumvent not only threading conflicts, but other potential conflicts as well. A variety of experiment control features are incorporated in the proxy, and have many applications throughout the design of a fault injection campaign. The proxy also integrates a network fault injector using Linux IP Chains firewall administration; the injector can cause and repair network partitions.

The second half of the thesis presents an experimental evaluation of correlated network partitions on the Coda distributed file system, based on the ideas that were described in the first half of the thesis. The study is focused on the effects of correlated network partitions during a phase of Coda's directory resolution protocol. We set up the scenario by creating diverging replicas of a target volume during a network partition. When the partition was healed, a client requested that the divergences be resolved. During resolution of the target volume, a correlated network partition was injected during a targeted phase of Coda's directory resolution protocol. At the conclusion of experiments, Loki's analysis phase generated global timelines for each experiment, and Loki's measure estimation facilities extracted

useful results from the case study.

The experimental evaluation is presented in two parts. The first part gives an overview of the case study, including an overview of Coda and its directory resolution protocol, and the design details for the case study. The case study design uses two adjoining Loki state machines, one to track the state of a file server during Coda's resolution process, and a supplemental state machine that tracks experiment state. The design also includes fault injections and control modifiers that influence the execution of experiments. The second part of the experimental evaluation is a presentation of results. Three types of results were obtained: results pertaining to final resolution outcomes, temporarily unavailable resources, and end-to-end measures. Results showed that Coda's resolution techniques never incorrectly resolved the target volume. However, in some cases, a correlated network partition caused the volume resolution to produce a result requiring manual resolution, even though the partitioned updates in the experiment should have been automatically resolvable. Also, end-to-end resolution times from experiments involving correlated network partitions were 12 to 14 times larger than for the baseline case, due to time spent waiting for the previously locked target volume to unlock. Once the volume was lockable, Coda demonstrated the ability to quickly diagnose cases that it determined required manual resolution.

This thesis also makes a contribution by leading to several opportunities for future work, including Coda case study extensions, Loki improvements, and new Loki features. One way that the Coda case study could be extended is by obtaining measures other than those presented in this work. For instance, the end-to-end measures could be further broken down into state times for the different phases involved during a volume's resolution. Also, the time that subordinates sit idle during the resolution process could be quantified. Another extension would be to perform correlated network partitions into other phases of the directory resolution protocol. We did not do that in the original study because those states were not long enough to permit injection of the second network partition. It might be possible to solve this problem by extending the fault trigger for netpartition2 to include a larger group of states in the hope that the fault would inject in the intended state. Experiments containing incorrect injections could be filtered out of consideration using Loki's measures language. A third extension would be to inject correlated network partitions during file resolution rather than directory resolution. Another possible extension would be to incorporate the results from this thesis into an availability model for Coda to assist in speculation about dependability measures other than those computed from the experimental evaluation.

A second category of future work pertains to Loki improvements. An easy way to improve Loki is by extending its library of fault models. This could be done by implementing completely new fault types, and by extending the proxy's network fault injector. IP Chains

provides a vast array of packet-filtering options that could support faults such as packet drops, data corruption, and header corruption. Another useful improvement would be to add formalisms for specifying the state of the system under study. One new formalism could be that of multiple state machines per node. It could potentially allow a complex state machine to be broken into smaller, less complicated state machines that contain fewer states overall. (In order to represent the state space of multiple state machines with one state machine, the single state machine contains the cross product of the states involved in each of the smaller state machines. This can cause the state space to grow very rapidly.) Another improvement would be to extend or modify the existing measures language. As seen in Chapter 5, some results that are not difficult to explain in words can be quite cumbersome to define in the current measures language. Two possible extensions would be to provide regular expression support and to allow control flow manipulation between triples in a study-level measure.

A final category of future work would involve development of new Loki features. For example, it would be very useful if Loki could automatically generate state machines for a system under study. There are a number of ways such a feature might be implemented. For instance, state boundaries could be defined when communication takes place between processes involved in a distributed protocol. Another option would be to make state boundaries at different layers of a distributed protocol. Finally, the boundaries could be defined as the boundaries of method invocations.

REFERENCES

[1] M. R. Lyu, Ed., *Handbook of Software Reliability Engineering*, New York: McGraw-Hill, 1996.

[2] J. Arlat, M. Aguera, Y. Crouzet, J. Fabre, E. Martins, and D. Powell, "Experimental evaluation of the fault tolerance of an atomic multicast protocol," *IEEE Transactions on Reliability*, vol. 39, no. 4, pp. 455–467, October 1990.

[3] D. T. Stott, M.-C. Hsueh, G. Ries, and R. K. Iyer, "Dependability analysis of a commercial high-speed network," in *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, 1997, pp. 248–257.

[4] S. Dawson and F. Jahanian, "Probing and fault injection of dependable distributed protocols," *The Computer Journal*, vol. 38, no. 4, pp. 286–300, 1995.

[5] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A highly available file system for a distributed workstation environment," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 447–459, April 1990.

[6] P. Kumar, "Mitigating the effects of optimistic replication in a distributed file system," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, 1994.

[7] P. Kumar and M. Satyanarayanan, "Log-based directory resolution in the Coda file system," in *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, January 1993, pp. 202–213.

[8] P. Kumar and M. Satyanarayanan, "Flexible and safe resolution of file conflicts," in *Proceedings of the USENIX Winter 1995 Technical Conference*, 1995, pp. 95–106.

[9] B. Noble and M. Satyanarayanan, "An empirical study of a highly available file system," in *Proceedings of the 1994 ACM SIGMETRICS Conference*, May 1994, pp. 138–149.

[10] 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.

[11] 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.

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

[13] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer, "NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors," in *Proceedings of the 4th IEEE International Computer Performance and Dependability Symposium*, Mar. 2000, pp. 91–100.

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

[15] R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders, "Loki: A state-driven fault injector for distributed systems," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2000)*, June 2000, pp. 237–242.

[16] R. Chandra, M. Cukier, R. M. Lefever, and W. H. Sanders, "Dynamic node management and measure estimation in a state-driven fault injector," in *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, October 2000, pp. 248–257.

[17] K. R. Joshi, M. Cukier, and W. H. Sanders, "Experimental evaluation of the unavailability induced by a group membership protocol," in *Dependable Computing EDCC-4: Proceedings of the 4th European Dependable Computing Conference*, October 2002, 140-158.

[18] D. A. Henke, "Loki – an empirical evaluation tool for distributed systems: The experiment analysis framework," M.S. thesis, University of Illinois at Urbana-Champaign.

[19] K. R. Joshi, "Evaluating unavailability caused by group membership using global-state-based fault injection," M.S. thesis, University of Illinois at Urbana-Champaign.

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