EVALUATING UNAVAILABILITY CAUSED BY GROUP MEMBERSHIP USING GLOBAL-STATE-BASED FAULT INJECTION

BY

KAUSTUBH RAGHUNANDAN JOSHI

B.Eng., University of Poona, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2003

Urbana, Illinois

To my wife, parents, and family, for their love and support. To all my teachers, for giving me the most valuable gift in the world.

Acknowledgments

I feel fortunate to have had the opportunity to work with a talented and dedicated group of people in the PERFORM group. First and foremost, I would like to thank my advisor Prof. William H. Sanders, for supporting my endeavors and giving me the freedom to pursue my interests. He has been a constant source of encouragement, ideas, and advice throughout my time with the PERFORM group. I thank him for always having been accessible despite his incredibly hectic schedule, and for standing by me through thick and thin. His dedication and commitment to his work have always inspired me. I would also like to thank Dr. Michel Cukier, who, during his time with the PERFORM group co-advised me on my work. I thank him for his insight and advice, and for ensuring that I did not stray off-course in my research pursuits. This work would not have been possible without him.

I thank Jenny Applequist for her tremendous help in editing my work. She has never once complained about the insanely short deadlines I have imposed on her, and her assistance will always be appreciated. Many thanks go to Ryan Lefever for the countless discussions we have had on fault injection and Loki, and also for being a great person to share an office with. It has been a pleasure to have known and worked with him on the Loki project. I also thank Ramesh Chandra for his work on the Loki project.

I would like to thank the funding agencies that have financially supported my research. This material is based upon work supported by DARPA under grant 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.

The members of the PERFORM group have always provided me with an atmosphere that was both stimulating and enjoyable. I thank Sudha Krishnamurthy for her advice about graduate school, and for the long technical discussions that have helped clarify my thinking. I thank James Lyons, Vinh Lam, Salem Derisavi, David Daly, Hari Ramasamy, Sankalp Singh, Vishu Gupta, Prashant Pandey and the other members of the PERFORM group for their friendship, and for making my stay here enjoyable. Many thanks also go to some unnamed PERFORM members for introducing me to the world of Starcraft, and for

many evenings of intense online battles that I have enjoyed tremendously.

I thank my parents and Babytai for the encouragement they have given me over the years, and for their constant support. I thank my father for teaching me that if something is worth doing, it is worth doing well. And last but not the least, I am thankful to my wife Manisha for constantly supporting and encouraging me through ups and downs, and for sharing my life. This thesis would not have been possible without her support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The fault-tolerant and decentralized properties of distributed systems have made them ideal candidates for use in modern highly dependable applications. Indeed, the number of distributed systems being used for mission-critical applications is on the rise. However, even though fault tolerance is inherent in the nature of distributed systems, in practice, it is exceptionally difficult to get right. This has led to the emergence of the group communication paradigm. Group Communication [Bir96] is a service that provides primitives for reliably sending messages to sets of processes without requiring knowledge of the current membership of the set. Group Communication Systems typically are software components that implement commonly used group services (usually in the form of a communications stack) and allow distributed applications that use these services to be written on top. Although not a new concept [Bir85], group communication is gaining importance as a paradigm for structuring distributed systems because of the increasingly complex and sophisticated nature of modern distributed systems, as well as the increasing levels of dependability expected from them. Group communication systems guarantee certain dependability properties almost transparently, thus allowing designers to concentrate on application logic.

Dependable system designers often use the dependability-related properties of group communication systems (GCS) to make claims about the dependability of their own applications. This makes the verification and assessment of these properties an important endeavor. Consequently, formal specification and verification of GCS properties is a very active area of research [VKCD99, FLS01, Nei96, HLvR99]. Work has also been done on automated construction of such specifications from source code [KHH98]. These techniques and the properties they verify are primarily functional in nature; that is, they determine whether a particular algorithm and its implementation maintain certain invariants.

However, non-functional properties of Group Communication Systems (such as performance), which verification techniques neither model nor analyze, can also have a significant

effect on the system's dependability properties. For example, applications that require a virtual synchrony model [Bir96] usually force the GCS to block communication when membership changes occur. This blockage can cause a loss of availability of some or all components in the system, thus reducing its overall dependability [Lap95]. Thus the performance of the group membership component has a direct impact on dependability. The degree of this impact depends not only on the protocols and algorithms used, but also on the actual implementation. This observation leads to the necessity of experimental evaluation of the dependability-related performance characteristics of group membership protocols, to complement formal verification. This thesis describes an attempt to experimentally evaluate the performance of certain GCS components (particularly the group membership mechanism) and to demonstrate the impact of performance on the availability (which is a component of dependability) of the system. The goal is twofold. First, the evaluation must obtain quantitative conditional measures on how factors such as workload, group size, and times of fault occurrence affect availability. Second, enough light must be shed on protocol operation to allow enhancement of the protocols and removal of bottlenecks in the system. To perform such an evaluation, this thesis develops a novel approach to fault injection based on the global state of a distributed system.

1.1 Evaluation of Group Communication Systems

Group Communication Systems typically consist of several protocols, two sets of which are particularly important. The first set consists of the message ordering and delivery protocols, which ensure reliable delivery amongst the members of a group. They typically also ensure some sort of message ordering, like FIFO or total ordering amongst messages from different senders. The set of protocols is exercised during normal operation of the group, and its performance is measured by metrics such as throughput of messages or message latency. The second important set of protocols relates to maintaining group membership, i.e., a view of which nodes are in the group that is consistent across all group members. This set of protocols is exercised when some "abnormal" event occurs (i.e., a node fails, or a new node joins the group). Performance of these protocols is related to the percentage of abnormal events they catch (coverage) and how long it takes to handle these events. When coupled with group properties such as virtual synchrony, membership protocols often cause the group to block message transmission and receipt in the group for certain intervals of time. Hence, an especially important performance metric for membership protocols is the *blocking time*, which is defined as the time during which no nodes can send or receive messages. This

constitutes a loss of availability from the perspective of the nodes. Blocking time is usually a fraction of the total operation time of a group membership protocol. Note that blocking time is a distributed measure and cannot be computed from measurements on a single node alone.

Although there has been some work on evaluation of group communication protocols [ML98, UMK97], the primary focus has been on performance of message ordering and delivery protocols. In this thesis, an experimental evaluation of the blocking behavior of a widely used group membership protocol is presented that sheds light on how the blocking time, and hence the availability of the system, is affected under varying operating conditions. To study loss of availability due to blocking in a group membership protocol, it is convenient to treat events that cause membership changes in the system as faults, since they cause the system to go into an undesired (blocking) state. Then, the group membership protocol is the fault tolerance mechanism for membership change "faults." Consequently, the technique of fault injection is used to exercise the membership protocol and subsequently perform distributed measurements on the system to evaluate blocking time.

1.2 Fault Injection

Fault injection, described in [AAA+90] is defined as a technique to test a system with respect to a particular class of inputs, i.e., the faults. A tool called a *fault injector* is usually used to conduct experiments and to inject faults into the system under test. The results of fault injection experiments have been traditionally used to perform fault removal and validation of fault tolerance mechanisms. The faults injected are selected according to a "fault model" and are usually representative of the faults that are expected to occur in real life in the system under test. Many fault models have been studied in the literature and include faults ranging from simple bit flips [SSK+01] to manipulation of communications messages [DJ95]. "Fault triggers" are used to specify when a fault is to be injected into the system.

A large body of work exists on both fault injectors for networked systems [SFB+00, DJMT96, EL92, HSR95] and fault injection, including some work on fault injection of group communication systems [AAC+90, DJ95]. However, in the past, most fault injection efforts have concentrated either on assessing such statistical dependability metrics as coverage of fault tolerance mechanisms [AAC+90, SHRI97, CPA99], or on performing fault removal [DJ95]. The fault triggers that have been considered have ranged significantly. Coverage-related injections have typically used random triggers [SSK+01] (i.e., injecting faults at random places in a program), triggers based on execution profile (i.e., code execut-

ing more frequently is targeted with a higher probability), or triggers based on load (faults are triggered when system or network load exceeds a certain threshold). Injections done for the purpose of fault removal have used local state-based triggers [DJMT96]. State-based triggers allow very precise control over the conditions under which fault injection is done.

For the purposes of assessment of unconditional measures such as availability, random triggers are often suitable, because it can be argued that in real life, faults often occur randomly. However, assessing unconditional measures requires that system workload and conditions be representative of real life. This is often not possible due to lack of prior knowledge about representative workloads and conditions. Moreover, unconditional measures give little insight into how the system can be improved. Hence, it may be preferable in some cases to compute measures that are conditional on workload and/or other system parameters. If desired, unconditional measures can be computed from the conditional measures using a workload model. Computing conditional measures is very similar to fault removal, because means are required to coerce the system such that very specific conditions required for fault injection are met. This calls for precise control over injection as is offered by state-based triggers. However, the distributed nature of a group membership protocol suggests that conditions in which faults are injected may not depend only on the state of a single node in the system, but rather on the cumulative state of all nodes in the system. This cumulative state is called the "global state" of the system. Fault injection based on the global state of a distributed system is a challenging problem and has not been addressed by past work on fault injection.

Another challenging problem in evaluating the availability characteristics of a group membership mechanism is the kind of measures required. Traditional fault injections have used simple measures such as the percentage of experiments that fail in a particular failure mode [SSK+01], that are insufficient here. The measurements and measures required are time-oriented measures of the entire group, not just of a particular node. The topic of distributed system measurement and performance evaluation has been considered in the literature [BJS+95, LKG92]. However, the combination of fault injection coupled with distributed measurement to perform assessment of dependability characteristics in a distributed system implementation has not been previously addressed in the literature.

To solve both problems, a novel global-state-based fault injection technique is used in this thesis. The technique involves abstracting the operation of each node in the group with a state machine. A global state for the entire system is defined in terms of the local states of the nodes. Fault triggers are expressed in terms of this global state. Measurements of local state changes are taken locally, and are combined using an offline clock synchronization algorithm to form a timeline of global state changes. Measures are defined on this "global

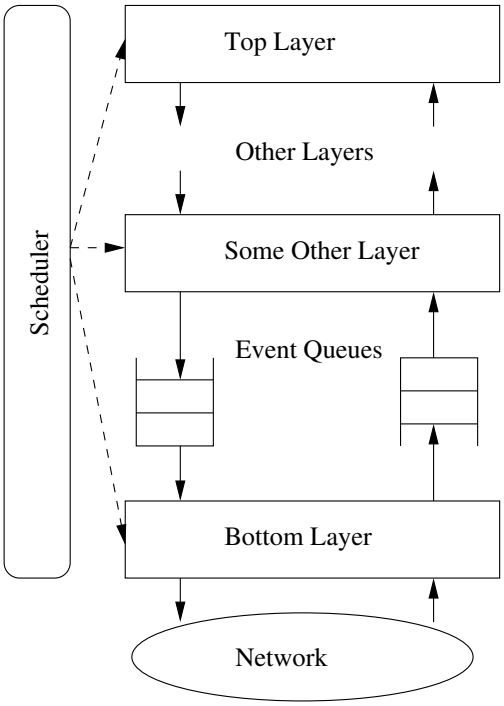
timeline." Loki [CLCS00], a fault injector that supports global-state-based fault triggers and measure estimation, is used to conduct experiments. This approach enables the injection of faults precisely under very specific conditions, collection of fine-grained measurements, and the subsequent computation of conditional measures of the system.

Although the approach presented here could be used to evaluate any group membership protocol, a single implementation was selected to perform fault injection and measurement on. The specific one chosen for the purposes of this study is the group membership protocol of the Ensemble GCS [vRBH+98]. One of the reasons for choosing Ensemble is that it is one of the most widely used group communication systems. Hence, it is well-supported, and is expected to be relatively free of bugs. Moreover, extensive work has been done in formally verifying the correctness of the protocols in Ensemble [HLvR99, KHH98]. Hence, any conclusions and findings resulting from an experimental evaluation of Ensemble support the claim that experimental fault-injection-based evaluation is a useful technique that is complementary to formal verification for high dependability systems.

1.3 The Ensemble Group Communication System

Ensemble is a popular group communication system developed at Cornell University. It is written in the OCAML dialect of the ML language to be amenable to automated proof checking [Hay97c]. Ensemble communication stacks have a modular structure and are built up of several layers of micro-protocols stacked on each other. Each micro-protocol layer implements (either completely, or in conjunction with other layers) a small piece of functionality called a "property." A *property* is some attribute, such as atomicity or total ordering amongst messages, that a group should possess. Ensemble defines several properties that include several types of message ordering (total order, causal order, or FIFO), group membership, virtual synchrony, and various types of security, amongst others. Several layers may be needed to implement some complex property. Conversely, there also may be several vastly different implementations of a single property (such as token-ring-based total ordering or sequencer-based total ordering), each with its corresponding set of micro-protocol layers. Ensemble allows an application to choose which micro-protocol layers to include in a communication stack, thus allowing applications to choose group properties. The micro-protocol layers work hand-in-hand using a well-defined inter-layer communication protocol [Hay97c] to implement the overall functionality required from the stack.

Figure 1.1(a) shows the structure of an Ensemble stack. An Ensemble communication stack is a highly layered entity composed of many (typically 15-20) layers. Inter-layer com-



(a) Structure of an Ensemble Stack

Layers	Functions	Properties
Top	Coordinate stack activities	
Heal	Find partitions to merge	Partition Merges (heal)
Leave	Handle node exits	
Inter	Handle partition merges	Group Membership (gmp)
Intra	Installation of a new view	
Elect	Elect new leader	
Merge	Request partition merges	
Slander	Share suspicions	Suspicion Sharing (slander)
Sync	Handle group blocking	View Syncrhonization (sync)
Suspect	Heartbeat failure detection	Heartbeats (suspect)
Stable	Msg stability detection	
Vsync	Virtual synchrony	
Partial_appl	Application interface	
Sequencer	Total ordering protocol	Total Ordering (total)
Collect	Maintain msg delivery info	
Frag	Msg fragmentation support	Msg Frag/Defrag (frag)
Pt2ptw	Unicast flow control	
Mflow	Multicast flow control	Flow Control (flow)
Pt2pt	Unicast reliable delivery	
Mnak	Multicast reliable delivery	
Bottom	Network iface	

(b) Stack Used for Evaluation

Figure 1.1: The Structure and Composition of an Ensemble Stack

munication is achieved solely via events between adjacent layers. If a layer receives an event it doesn't process, it simply passes it on. Consequently, only the bottom most layer can communicate with the network. Messages received from the application or the network are encapsulated in events before being manipulated by the stack. Each layer processes only one event at a time, and does so using a FIFO scheduling discipline. However, the system is free to schedule events in multiple layers concurrently. For a detailed description of the Ensemble layering model, refer to [Hay97c]. Since layers in a stack are scheduled independently, there is no order among layer executions. In addition, several events can exist concurrently in the stack. This greatly complicates the problem of representing the temporal behavior of the entire stack using state machines. This represented a significant challenge during the evaluation process, and its solution is presented in Section 3.4.

The Ensemble stack used during the evaluation was configured as shown in Figure 1.1(b). The figure shows the layers used, their functions, and the properties that led to that particular configuration of the layers. Briefly, the stack used for the evaluation possessed properties of sequencer-based total ordering, credit-based flow control, group membership, virtual synchrony, heartbeat-based crash detection, a gossip mechanism for crash-suspicion sharing, group merging mechanisms (to support network partition), and support for members volun-

tarily leaving or joining a group. This configuration (with the exception of total ordering) is the default configuration of the Ensemble distribution.

1.4 Contributions and Outline

The main contributions of this work are twofold. First, this thesis attempts to develop, through example and otherwise, the methodology of global state-based fault injection, and is one of the first attempts to use this technique to perform system evaluation. Second, this is one of the first attempts to systematically assess and quantify the dependability properties of a group membership protocol through experiment. Both the quantification of the cost of group membership operations, and the insight gained in this process, form an important contribution.

Therefore, this thesis is divided into two main parts. The first part consists of Chapter 2, and describes the fault injection methodology and the associated fault injector Loki [Cha00], which was used to implement it. Chapter 2 also describes the problems encountered during fault injection that were solved through algorithmic or other enhancements to Loki. Finally, the chapter presents experimental results that evaluate the precision, accuracy, and intrusiveness of the Loki fault injector implementation and validate some assumptions on which that injector is based. These experiments help us understand the extent to which results obtained using Loki are valid, and help the reader gain some perspective into some of the limitations of the implementation.

The second major part of this thesis consists of Chapters 3 and 4, and presents the evaluation of the Ensemble group membership protocol. Chapter 3 describes the technique used to construct state machine models of a highly layered event driven-entity such as an Ensemble stack, and the resulting state machines for the Ensemble group membership protocol. These state machine models form the basis for the global state-based injection. The fault definitions for the injections and measures used to evaluate the group membership protocol are also presented in this chapter. Chapter 4 presents the results of the evaluation, describes the relevance of the results to the availability of an Ensemble-based system, and presents the insights gained that can help improvement of the protocols. Our insights and experiences regarding the fault injection process are also described here. Finally, Chapter 5 concludes with some ideas for future work.

Chapter 2

The Loki Fault Injector

This chapter describes Loki [CCH+99, CLCS00, CCLS00, CLJ+02, JCS02], the fault injector that is used for our experimental evaluation. Section 2.1 describes the philosophy behind Loki and how this philosophy has shaped Loki's fault injection process. Section 2.2 describes the enhancements that we made to the fault injector as a result of our work. Section 2.3 presents a brief description of the measures implementation that was done as a part of our work. Finally, Section 2.4 presents an experimental evaluation of the Loki implementation in an attempt to provide insight into the kinds of fault injections that can be done using Loki and to quantify the accuracy of the measures.

2.1 Overview

Fault Injectors have usually been categorized according to three properties. The first is their target application domain, i.e., whether the intended target of fault injection is a piece of hardware, a simulation, or a software implementation. The second property is the supported fault model, i.e., the kind of faults (such as bit flips or message corruption) the fault injector can inject into a system. The final property by which fault injectors are usually categorized is the type of triggers that can be used to specify when faults are to be injected into the target. This categorization to a large extent determines how a fault injector is used and what kinds of experiments can be performed using it. In addition to the above three fault injector properties, Loki adds the new property of measures languages. A measures language determines the suitability of a fault injector for an experimental study, because it defines what measurements can ultimately come out of the fault injection process. The rest of this section is devoted to describing Loki in terms of the above properties and throwing light on how a fault injection campaign can be carried out using Loki.

2.1.1 System Model

Loki is a fault injection tool for distributed system implementations. In order to understand the fault triggers and measures supported by Loki, it is first necessary to understand how Loki models a distributed system. A distributed system is assumed to consist of a finite set of software entities (referred to as *nodes*) distributed on a set of host computers. A fault injection experiment begins with an initial set of executing nodes. Non-executing nodes can begin execution, or currently executing nodes can terminate, at any time during the execution of the experiment. An experiment is assumed to be complete when no executing nodes are left in the entire system. From the point of view of the system model, the definition of a node is left imprecise and could be taken to mean any software entity such as a process, a thread, or some other abstraction that is appropriate for the target application. However, the current implementation of Loki equates nodes with processes.

Loki models each node in the system as a finite state machine (FSM) and the entire distributed system as a collection of FSMs. The state of a node in the system at any instant of time during an experiment execution can then be represented solely by the state of its FSM at that time. This state is called the *local state* of the node. The local states of nodes that have not yet begun execution or that have finished execution are represented by special `BEGIN` and `EXIT` states in their FSMs. Changes to the local state of a node are caused by events occurring within the software entity that the node represents. These events are indicated by transitions in the node's FSM and are termed *local events*. The vector at any instant of time of the local states of all the nodes in a distributed system is called the *global state* of the system at that instant of time. Changes in global state are caused due to changes in one or more constituent local states, which are in turn caused by local events. Hence, the execution of an experiment is completely specified by an initial global state at the beginning of the experiment, and a sequence of pairs of local events and their occurrence times according to a common clock. This sequence is called a *global timeline* and effectively represents a global execution trace of the distributed system for the duration of the experiment.

An important point to note is that local events are measured in terms of local times on the hosts on which they occur. Converting these local times into a global time as required for the global timeline represents a significant challenge. Loki utilizes an offline clock synchronization algorithm (described in [Hen98]) to perform this conversion. This clock synchronization algorithm can place deterministic global time bounds on a local event occurrence. This means that the global timelines that are actually generated are not ideal because they contain sequences of event occurrence *intervals* instead of sequences of event occurrence times. This is shown in Figure 2.1, which shows how local event occurrence times are combined to form

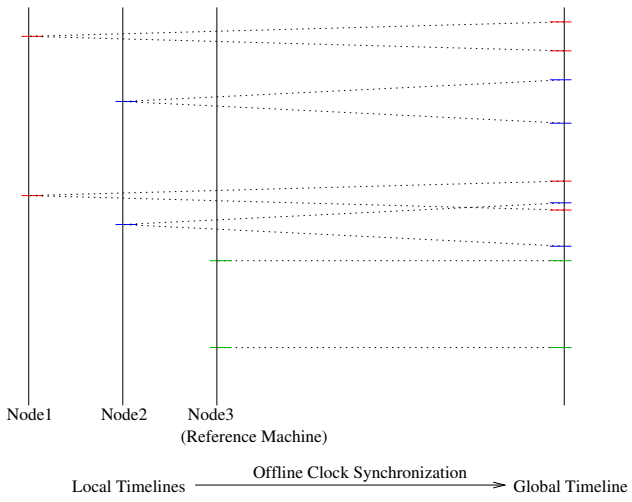


Figure 2.1: Global Timeline Generation

a global timeline for an experiment execution. The lengths of these intervals depend on several factors (which are described in [Hen98]) and are quantified for a typical case in Section 2.4. However, there always exists a reference host (usually the fastest machine) whose clock is used as the "global time." The precise timing for events occurring on this host is known, and the length of the event intervals for this host is zero. The fact that the global timelines contain event occurrence intervals instead of event occurrence times has important implications for the verification of the correctness of fault injections and measure computation. This is discussed in Section 2.2.1.

Experiments conducted using Loki are categorized into campaigns and studies. A *study* is defined as a set of similar experiments (i.e., using the same system specification, triggers, and fault model) conducted for the purpose of obtaining statistical measures. A *campaign* is defined as a set of related studies. Loki provides ways to combine measures obtained on different studies in a campaign, and hence it is desired that the user group studies representing the various parts of an overall measure into a single campaign. Each study in a campaign may have different specifications, triggers, or fault models, making it possible to combine studies representing different measures of a system, or even different systems.

2.1.2 Triggers and Fault Model

Loki allows injection of faults into nodes of the system based on the notion of global state. In Loki, a fault is specified completely by its name, the name of the node in which it is to be injected, and a fault trigger that specifies the exact conditions under which this fault is to be injected. The fault trigger is specified as a Boolean expression of global state queries.

A *global state query* is written as (Node_n : State_s) and returns *True* whenever Node_n is in State_s. For example, the trigger (Leader : Phase1) $\wedge$ (Follower : Phase2) $\mapsto$ MyFault indicates that when the node Leader is in the state `Phase1` and the node Follower is in the state `Phase2`, the fault `MyFault` should be injected into the node on which the trigger is defined.

An important point to note is that as far as Loki is concerned, the fault is identified only by its name. There is no reference to the fault model or fault type. The reason is that Loki leaves the implementation of the fault entirely up to the user. The Loki user implements the required fault models and makes them accessible through an "injection routine." When a fault trigger is activated, Loki invokes the injection routine with the name of the activated trigger. The injection routine is then responsible for matching the name of the triggered fault to the appropriate fault model and actually injecting the fault. In this respect, Loki can be seen as a triggering mechanism based on global state, and conceptually, any fault model for which an injection method can be written by the user is supported under Loki.

2.1.3 Measures

Loki allows, via a measures language, the definition of measures on the global timelines that result from experiment executions. There are two types of Loki measures: study-level measures and campaign-level measures. A *Study-Level Measure* operates on the global timelines generated by each experiment in a study and generates a single numerical value, which is the value of the measure, per study. A *Campaign-Level Measure* is a tool that allows the computation of statistical features such as moments and alpha levels for the population of measure values formed by either a single study-level measure or a meaningful combination of multiple study-level measures. In this thesis, the word "measure" is taken to mean a study-level measure, unless otherwise qualified.

A study-level measure is a multi-tiered structure. Each tier consists of the following three functions. The syntax for these functions is described in detail in [Cha00].

The predicate function is a selection function that operates on an experiment's global timeline. This function selects portions of the global timeline that are interesting and generates a Boolean-valued function of time called the *predicate timeline*. The predicate function is defined as a Boolean expression of operators that query the global timeline for certain global states or events.

The observation function is an aggregation function that operates on the predicate timeline. This function aggregates information contained in the predicate timeline into a

single numerical value called the *observation value*. The observation function can be any legal C code that returns a numerical value. Several aggregation functions are provided to query attributes of the predicate timeline. Examples include functions that measure how long a predicate timeline is true or false, and functions that count the number of Up and Down transitions of the predicate timeline.

The subset selection function uses the observation value generated by the the observation function and standard comparison operators to determine whether to filter out the current experiment, or to forward it to the next tier.

An experiment starts from the top, and passes through the tiers one by one. If it passes through all the tiers, then the observation value of the last tier is the value of the study-level measure for that experiment. If the experiment is rejected at some tier, then it does not generate a value for the study-level measure. Note that in such a structure, all of the tiers but the last one act as filters and do not contribute to the actual computation of the final measure value. The reason is that observation values for a tier are discarded when the experiment passes to the next tier. This has implications for the expressiveness of the measures language. This issue is dealt with in Section 2.2.2.

Since a study-level measure generates a single measure value for each experiment in a study, each of these measures generates a population of values. A campaign level measure is defined on one or more study-level measures. The campaign-level measure computes statistical characteristics such as moments, skewness and kurtosis coefficients, and alpha levels of the combined population on which it is defined. Since different study-level measures in a campaign-level measure may mean different things, study-level measure populations can be combined in different ways depending on the application. Campaign-level measures are differentiated based on how the populations are combined, as follows.

Simple Sampling measures treat all the input populations as if they were generated from the same source.

Stratified Weighted measures assign weights to the different input populations, and compute the statistical features for a source that is equivalent to a weighted sum of the sources represented by each input population.

A Stratified User measure allows the user to combine the means of the input populations in an arbitrary manner to get the resulting measure. Statistically, such a combination may not have a well-defined meaning. Hence, computation of statistical features for this kind of measure is not supported.

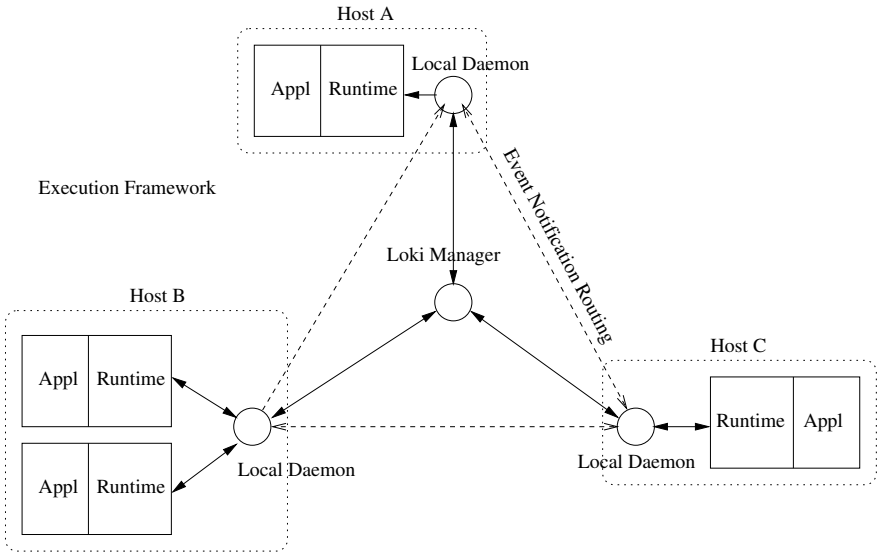


Figure 2.2: Loki Architecture

2.1.4 Loki Implementation

The architecture of the Loki implementation has a bearing on the target application domain and the fault models that can be supported by Loki. Figure 2.2 shows the architecture of the Loki implementation. As seen in the figure, the implementation consists of two parts: a manager and an execution framework. A brief description of the manager and the runtime is presented here. Additional details can be found in [Cha00].

The Loki Manager is responsible for the tasks of study specification, experiment execution, and measure computation. The manager consists of a user interface that allows the Loki user to specify studies and measures. The interface also allows the user to execute a study, analyze a study, and compute measures. Another part of the manager, called the central daemon, oversees study execution by starting up the execution framework at the beginning of a study, starting up initial application nodes at the start of each experiment, monitoring the execution of the experiment, shutting down the application nodes when the experiment finishes, and shutting down the execution framework when the study finishes. The central daemon also manages the collection and archival of log files from the experiment.

The Loki execution framework consists of two components: the local daemons and the Loki runtimes. During a study execution, one local daemon runs on each host in the system. A local daemon performs several functions, including starting up application nodes on the host, routing event notification messages between local and remote nodes, detecting whether a local node has terminated or crashed by using a combination of heartbeats and

explicit notifications, and finally, interacting with the central daemon to manage experiment execution.

The Loki runtime, which is a component of the execution framework, must be attached to each node (process) of the distributed system. The runtime is responsible for keeping track of the experiment execution in terms of the system model described in Section 2.1.1. During the course of an application execution, the runtime tracks local events occurring in the node it is attached to, records the event changes, and communicates them to other nodes when necessary. Each runtime also maintains a *partial view* of global state [Cha00], which is that subset of the current global state of the system needed for evaluating fault triggers. The runtime at each node uses its partial view of global state to determine when fault triggers fire, and when they do, the runtime injects faults into the node. Since application execution is not stopped while a event is being recorded and fault triggers are being evaluated, fault injection is optimistic. This means that there is a chance that by the time a fault expression is evaluated and the fault is injected, the system may no longer be in a global state in which the fault trigger is true. This can happen if nodes change state quickly enough.

The runtime consists of a number of threads that execute inside the application process along with any other application threads. To allow the runtime to perform its functions, the application needs to be linked with a runtime library provided by Loki, and the application startup code needs to be instrumented to start the threads up. Whenever a local event that changes local state occurs during the execution of the process, the application needs to notify the runtime of the event. It does so by calling a `notifyEvent` method that is exported by the runtime library in the form of a C function call. In addition, the fault models must also be implemented by the user of Loki (i.e., the person performing the fault injection). He or she does so by writing an *injection method*, which is a C function called by the Loki runtime whenever a fault is to be injected into the node. The runtime passes the injection method the name of the fault to be injected. It is the job of this method to implement the fault model based on the fault name that was passed to it. Hence, it is theoretically possible to use Loki with any distributed application for which events can be monitored and passed to the Loki runtime via a C interface, and the fault models can be implemented by the user as a combination of hardware and software. However, the applicability may be limited by the precision of fault injection and the event detection granularity offered by the Loki implementation (Section 2.4).

In practice, node events can be captured and faults injected into the system in a variety of ways, each of which affects the granularity and precision differently. One possibility is to instrument the application code in order to attach the runtime to it. Subsequently, events can be passed to the runtime using application mechanisms and a C interface. This method

gives the best precision and granularity, but can be intrusive to the application. It requires that a C interface be available to the application, and the application's threading model be compatible with the model used by the runtime. Another possibility is to use an external monitor to detect events based on the application's external output interface, and to inject faults using its external input interface. This approach is the least intrusive one, but limits the kinds of events/faults that can be detected/injected. Also, the precision and granularity of this approach are expected to be worse than those of the previous approach. This approach was explored in [Lef03]. Since Ensemble satisfies the requirements of threading model compatibility needed for the first approach, our work uses that approach for experimentation. However, the Ensemble GCS is written in the ML language, and a C interface is needed for interfacing with the runtime. Appendix B describes the C/ML interface that we wrote for that purpose. Section 2.4 quantifies the granularity, precision, and other performance measures of the Loki runtime when used with the first approach.

2.2 Loki Enhancements

A substantial part of the Loki system model and architecture as described in Section 2.1 was presented in [Cha00]. However, Loki, and in general the concept of global-state-based fault triggering, have never before been used for the fault injection of a complex and widely used distributed system. During the experimentation presented in this work, the complexity of the system under evaluation and the inherent challenges encountered during the process of global-state-based fault injection led to several enhancements to the Loki architecture and to the analysis and measurement process. Several extensions to the Loki measures language were also done in an attempt to either enhance the expressive power of the language, or simplify the task of defining measures. Some of the most important enhancements are presented in this section. The addition of named tuples and support for incorrect injection removal enhance the expressive power of the language, while the support for higher-order predicates increases conciseness of the language.

2.2.1 Identification of Incorrect Fault Injections

Loki fault injection is optimistic in the sense that faults can be injected into the system even when the system is no longer in the state that triggered the fault. This means that to ensure correct experimental results, the validity of injected faults must be checked before measure computation. In the analysis phase, after the global timelines have been generated, Loki performs an offline analysis to determine if all faults in an experiment were injected,

and if they were, whether they were injected in the correct global states. Each fault is then appropriately labeled as `NOT INJECTED`, `CORRECT` or `INCORRECT`. In the following discussion, the algorithm used to perform this check is described.

Given a global timeline and a fault trigger expression, the task of the injection validation algorithm is to determine whether or not the state of the system (as described by the global timeline) satisfies the trigger expression at the time of the fault injection. Since global state changes are a result of local events on the timeline, the global state at any point on the timeline can be computed by going through each event in ascending order of time, and updating the global state whenever local events occur. However, in reality, the situation is complicated by the fact that events on the global timeline are not defined at precise instants, but over an *interval* of time. Hence, between the upper and lower bounds of an event occurrence interval, the state of the system is indeterminate. Consequently, the global state of a system over the duration of an experiment can be thought of as consisting of long periods of certainty (the precise global state is known) with short periods of uncertainty (associated with local event transitions) interspersed among them; the less precise the clock synchronization, the longer the length of these intervals of uncertainty. Due to the nature of the clock synchronization algorithm used by Loki, it can be proved that the uncertainty intervals are lower-bounded by the minimum round-trip delay of the network [CLJ+02]. Section 2.4 presents the lengths of typical uncertainty intervals associated with each event for the type of experimental setup used in this study.

The uncertainty associated with the global timelines makes a precise knowledge of global state at any instant impossible. A straightforward technique to circumvent this uncertainty problem is to add an additional condition that must be met for a fault injection to be labeled as correct. The condition is that during the entire interval occupied by the fault injection event, the state of the system must be determinate, and be such that it satisfies the fault trigger condition. This condition ensures that only correct fault injections are labeled as such. However, it suffers from a false negative problem in that some fault injections that were correct may be labeled incorrectly. This can happen when a fault injection interval overlaps with the uncertainty interval for an event, but the old global state and the new global state (as a result of the event occurrence) both satisfy the fault trigger. Figure 2.3 shows two situations in which this happens. The first example shows a situation in which there is a self loop in the state machine. The second example demonstrates the false negative problem when a fault trigger tries to aggregate states together for the purpose of injection. The second example is disturbing, because such aggregation is sometimes the only way to inject faults in very short-lived states, and the false negative problem can cause mislabeling of such injections.

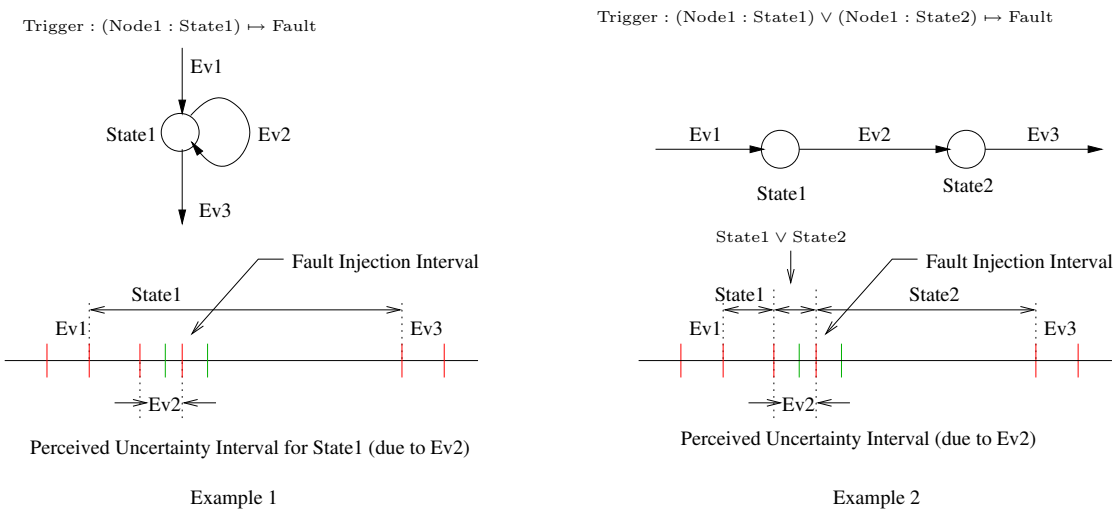


Figure 2.3: Examples of Incorrect Fault Labeling

The false negative problem is removed by extending the simple solution described above such that it keeps track of the source of indeterminism at all times. A key observation is that indeterminism on the global timeline is caused by imprecise timing information of local events that occur in a certain order. During the uncertainty interval for a local event, the node on which the local event occurred can be in either the old state or the new state. If the uncertainty intervals for two events occurring on the *same node* overlap, then the node may be in one of three states. In general, if the uncertainty intervals for n local events overlap each other, then the node may be in one of $n + 1$ states. Figure 2.4 shows an example with several such overlaps. Hence, the *local state cover* for a node at an instant of time is defined as the set of states in which the node could possibly be at that time. If the cardinality of the local state cover is 1 for a period of time, then it represents a period of certainty. The *global state cover* for the system can then be defined as a vector of local state cover sets, one for each node. The set of possible global states a system can be in at any instant of time is given by the Cartesian product of all the local state covers in a global state cover. If the fault trigger for a fault is satisfied by all the possible global states the system could be in during the interval of a fault injection event, then the fault is injected correctly.

The computation of the global state cover is simplified by the observation that uncertainty intervals for a single node increase monotonically with time [CLJ+02] (see Section 2.4 for experimental validation of this phenomenon). Hence, an uncertainty interval can never completely fall within another uncertainty interval for the same node. This implies that a strict FIFO ordering holds for the entry and exit of node states into and from the local state cover. Hence, it is possible to compute the global state cover at any instant by traversing the

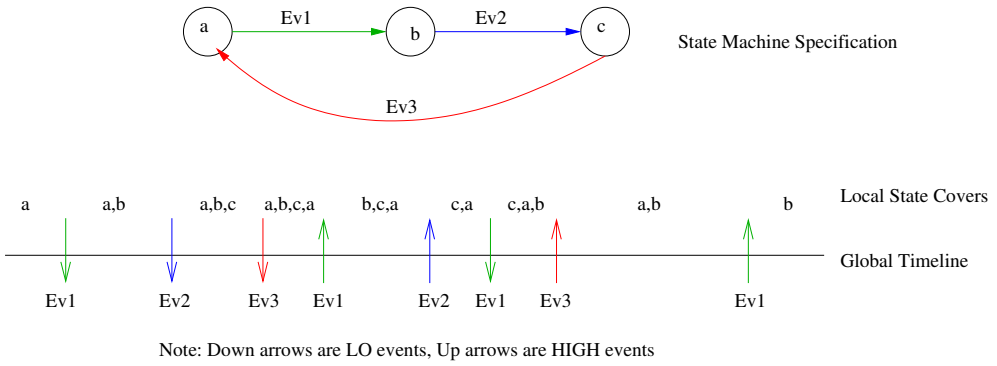


Figure 2.4: Computation of Local State Covers

global timeline in ascending order of time. Whenever the lower bound of a local event occurs, the new state resulting from that event is enqueued in the node's local state cover. Whenever a higher bound for the event occurs, the old state for that event is at the head of the queue due to the FIFO property, and hence it is dequeued. The computational complexity of the operation is linear with respect to the number of events in the global timeline. However, computing whether a fault trigger can or cannot be satisfied by a global state cover is not so computationally simple. In fact, it can be proven that this operation is NP-Complete.

To prove NP-Completeness, the problem `global-state-cover-satisfiability` (or GSC-SAT) is formulated, which, when given a global state cover and a fault expression, returns True when even a single global state in the global state cover does not satisfy the fault expression. Therefore, faults can be labeled as incorrectly injected if GSC-SAT returns True anytime during the fault interval. The first part of the proof of NP-Completeness [CLR90] must prove that GSC-SAT belongs to the class NP, i.e., it is verifiable in polynomial time. To prove polynomial time verifiability, it must be shown that if a fault trigger is not satisfiable by a global state cover, then there exists a certificate to prove that it is not. It should take at-most polynomial time to verify this certificate. The certificate can be obtained as follows. If a fault trigger is not satisfiable by a global state cover, there is at-least one global state in the cover that does not satisfy the trigger. This global state can be used as the certificate. It is possible to check the validity of the global state certificate by evaluating the fault trigger with the corresponding global state; the evaluation is a polynomial time operation. Hence, GSC-SAT belongs to the class NP. Further, it must be proved that an existing NP-Complete problem reduces to the GSC-SAT problem. Consider a global state cover in which each local state cover has just two states '0' and '1'. Then, the problem of determining if the cover satisfies a fault trigger is equivalent to solving the Boolean satisfiability problem for that fault trigger expression. Hence, Boolean satisfiability reduces to GSC-SAT, thus proving

that GSC-SAT is NP-Complete. Using this result, it is clear that the overall problem of incorrect fault removal is NP-Complete as well.

Based on the preceding discussion, the algorithm for computing whether a set of faults F were injected correctly into the system is as shown in Algorithm 1. The inputs to the algorithm are an ordered set N of nodes, an ordered set E which specifies the local events for each node, an ordered set F of faults, and a global timeline G. The subscript operator is used to extract the elements of a vector. Hence, $N_i, i = 0, \ldots, |N|$ refers to the i^{th} node from the set N. With each node N_i is associated a set of events that can occur on the node ($E_i \in E$) and a set of states the node can be in (S_i). The global state of the system gs is a vector $gs \in GS$, where $GS = [s_0, s_1, \ldots, s_i, \ldots, s_{|N|}], s_i \in S_i$. Each fault $F_i, i = 0, \ldots, |F|$ in the fault set is associated with a fault trigger $T_i(gs)$. The trigger is a Boolean expression on the global state of the system $gs \in GS$. The function $\texttt{EVAL} : expr \times GS \mapsto Boolean$ accepts a Boolean expression on the set of global states, a global state $gs \in GS$, and returns the Boolean result of evaluating the expression on the given global state. The running time of the function is linear in the size of the expression (i.e., $O(|expr|)$). The global timeline G is an ordered list of tuples $(N_i, event, time, type)$. For each tuple in the timeline, $N_i \in N$ specifies a node, $event \in E_i$ specifies a local event, $time$ specifies the global time of occurrence, and $type \in \{\text{LO}, \text{HI}\}$ specifies whether $time$ in the tuple is the lower or upper bound of $event$, respectively. The global timeline is assumed to be sorted in the ascending order of $time$.

The algorithm operates by scanning through the global timeline, one event at a time. At each step, the algorithm keeps track of two sets of items. The first set of items are the local state covers for each node. These are maintained using a FIFO queue. The second set of items tracked at each step are the "Open Faults." *Open Faults* are those whose lower bound has been encountered during the global timeline processing, but whose upper bound has not. A fault injection is incorrect only if the global state cover, at any time while the fault is an open fault, contains a global state that does not satisfy the fault trigger. Computing the satisfiability of a fault trigger by a global state cover was earlier proved to be NP-Complete. Therefore, the algorithm simply performs the task by enumerating all the possible global states in the cover, and evaluating the trigger expression for each global state. In practice, the local covers for only those nodes on which a fault trigger expression depends are used in the computation of the global cover used to check that trigger. Let the fault trigger expression T_k for fault F_k query the local state of n_k distinct nodes. Let e_k be the number of events that occur, within the fault interval for F_k, on nodes that T_k depends on. Let lc_{max} be the maximum cardinality of a local state cover set. Then, the time complexity of checking for a correct fault injection of F_k can be bounded by $O(e_k \times (lc_{max})^{n_k})$. Hence, if there are $|G|$ events in a global timeline, and the number of faults to be checked is $|F|$, then Algorithm

Algorithm 1: Algorithm to Label Correct Fault Injections

Input:

$N = \{N_i | i = 1, \ldots, |N|\}$, an ordered set of nodes,
$E = \{E_i | i = 1, \ldots, |N|\}$, where E_i is the set of local events on node N_i,
$F = \{F_i | i = 1, \ldots, |F|\}$, an ordered set of faults,
$T = \{T_i | i = 1, \ldots, |F|\}$, an ordered set of fault triggers,
$G = \{(N_i, event, time, type)_p | p = 0 \ldots |G|, N_i \in N, event \in E_i,$
$type \in \{\text{LO}, \text{HI}\}\}$, a global timeline.

Output:

$\{\text{label}_i | i = 1, \ldots, |F|, \text{label}_i \in \{\text{CORRECT}, \text{INCORRECT}, \text{NOT\_INJECTED}\}\}$,
an ordered set of the injection status of each fault in F.

LABELFAULTS(N,E,F,T,G)

(1) **Array of Queue** $LocalCover[1 \ldots |N|]$
(2) **Array of Label** $InjectionStatus[1 \ldots |F|]$
(3) **Set** $OpenFaults \leftarrow \{\}$
(4) **for** $i \leftarrow 1$ **to** $|N|$
(5) ENQUEUE($LocalCover_i$, InitialState$_i$)
(6) **for** $k \leftarrow 1$ **to** $|F|$
(7) $InjectionStatus_k \leftarrow$ NOT_INJECTED
(8) /* Process each event in the global timeline */
(9) **for** $p \leftarrow 1$ **to** $|G|$
(10) $(N_i, event, time, type) \leftarrow G_p$
(11) **if** $event = F_k$ /* event is a fault */
(12) **if** $type =$ LO **and** $InjectionStatus_k \neq$ INCORRECT
(13) $OpenFaults \leftarrow OpenFaults \cup \{F_k\}$
(14) $InjectionStatus_k \leftarrow$ CORRECT
(15) **else if** $type =$ HI
(16) $OpenFaults \leftarrow OpenFaults - \{F_k\}$
(17) **else** /* event is a normal event */
(18) $ns_i \leftarrow$ NEXTSTATE($N_i, event$)
(19) **if** $type =$ LO **then** ENQUEUE($LocalCover_i$, ns_i)
(20) **else** DEQUEUE($LocalCover_i$)
(21) /* Test if the event invalidates any fault injection */
(22) **foreach** $F_k \in OpenFaults$
(23) **if** T_k depends on N_i
(24) **for** $j \leftarrow 1$ **to** $|N|$
(25) **if** T_k depends on N_j **then** $cover_j \leftarrow LocalCover_j$
(26) **else** $cover_j \leftarrow \{\text{InitialState}_j\}$
(27) $GlobalCover \leftarrow cover_1 \times \ldots \times cover_{|N|}$
(28) **foreach** $gs \in GlobalCover$
(29) **if** $\neg$EVAL(T_k, gs)
(30) $InjectionStatus_k \leftarrow$ INCORRECT
(31) $OpenFaults \leftarrow OpenFaults - \{F_k\}$
(32) **break**
(33) **return** $InjectionStatus$

1 takes $O(|G| + \sum_{k=1}^{|F|} e_k \times (lc_{max})^{n_k})$ time. Fortunately, the running time is manageable in practice because the values of e_k and lc_{max} are usually small. They are small because event intervals on the global timeline are usually a small constant times the minimum round trip latency of the network, and the rate of state changes in typical distributed systems is limited by synchronization delays, which are latency-bounded.

2.2.2 Named Tuples

In the measures language described in Section 2.1.3, tuples are arranged in an ordered sequence. An experiment passes through each tuple in the sequence in order. To reach a tuple, an experiment must pass through all preceding tuples. Additionally, information gleaned from the processing in one layer cannot be carried over into subsequent layers. Hence, all but the final layer serve only to filter the experiments, and computation of the result can be done only in the final tuple. Each tuple can evaluate only a single predicate timeline. While this simple filtering action of all but the last tuple is sufficient for simple measures such as state sojourn times, it is not enough for more complex measures. An example of such a complex measure is one that computes the fraction of a system's up-time that was spent doing error recovery, or one that measures the difference in rates of incoming events and outgoing events in a unstable system. In general, is is not possible to compute measures which need to combine two independent pieces of information from the global timeline (i.e., measures for which two independent predicate timelines are required). In the first of the above example measures, the independent pieces of information were the error recovery time and the up-time of the system, while in the second example measure, they were the incoming events and outgoing events.

In order to take care of the problem, named tuples were added to the language. The addition of named tuples simply means that each tuple is given a name that can be used to access the observation values of that tuple in subsequent tuples. Hence, as an experiment passes through the tuple chain, the observation values generated at each tuple are not discarded, but are stored in a variable that can be accessed by the tuple name. This implies that different tuples, each with its own predicate function, can be used to extract independent pieces of information from the global timeline, and that this information can be combined in subsequent tuples. Observation values from previous tuples can be used in any part of a tuple viz. the observation function, predicate function, and subset filter. Examples of how the measures described previously can be expressed with the new extension are shown in Table 2.1.

Table 2.1: New Measures Computable Using Named Tuples

Ratio of error-recovery time to uptime	
Tuple	UPTIME
Predicate Fn	chk_state(Node_1, Up) $\wedge$ chk_state(Node_2, Up)
Observation Fn	total_duration(TRUE, Exp_Begin_Time, Exp_End_Time)
Filter Fn	1
Tuple	FINALMEASURE
Predicate Fn	chk_state(Node_1, ErrorRecovery) $\wedge$ chk_state(Node_2, ErrorRecovery)
Observation Fn	total_duration(TRUE, Exp_Begin_Time, Exp_End_Time)/UPTIME
Filter Fn	1
Difference between input and output rates	
Tuple	OUTPUTRATE
Predicate Fn	chk_event(Node_1, OutputEvent)
Observation Fn	count(UP, IMPULSE)
Filter	1
Tuple	FINALMEASURE
Predicate Fn	chk_event(Node_1, InputEvent)
Observation Fn	count(UP, IMPULSE) - OUTPUTRATE
Filter	1

2.2.3 Higher-Order Logic

The predicate functions used in the Loki measures language are Boolean expressions that can query the state of a node in the system, the occurrence of events in the system and the status of fault injections (with the modifications from Section 2.2.4). Although Boolean functions provide the power needed for most useful measures, they are not scalable. When the number of states in the nodes or the number of nodes in the system becomes large, then useful predicate functions using the primitives of $\vee$, $\wedge$, and $\neg$ become too large to be practicable. As a tool to reduce the size of predicate functions, higher-order logic was introduced into the measures language. Strictly, it does not increase the expressive power of the language, but greatly simplifies the specification of complex predicates. Support for higher-order logic is based on "closures", which are structures commonly used in higher-order programming languages.

A *closure* is a structure that contains the body of an expression along with an "environment". The expression body typically contains free variables. The *environment* contains assignments of values to a subset of the free variables used in the expression body. A free variable in a closure can be "bound" by passing a value for that free variable to the closure. Binding a free variable to a value causes a mapping between the free variable and

the value to be added to the environment. A closure can be evaluated only when all the free variables in the expression body have been bound. Such a closure is referred to as *fully bound*. In the Loki measures language, a closure is written as $[x, \text{Expression using} x]$. For example, $[x, \text{chk\_state}(\text{Node}_1, x)]$ is a closure that when passed the name of a state (x), becomes fully bound, and returns True if Node_1 was in that state. In the measures language, each closure introduces exactly one free variable (x in the previous example), and always returns a Boolean value when evaluated. The expression body of a closure can also contain other nested closures, all of which may use the free variable introduced by the outer closure. When the free variable introduced by a closure is bound to a value, all the occurrences of that free variable in any nested closures are also bound to the same value. An inner closure can reintroduce a free variable that has already been introduced in an outer closure. In that case, the reintroduced free variable causes any previous binding for that variable to be overridden in the environments of the inner closure and any other closures nested within the inner closure (similar to the nested scope rules in programming languages). The process of binding the free variable introduced by a closure to a value is referred to as applying the closure to that value.

In addition to closures, we introduced second-order functions that provide the means for binding the free variables in a closure. More specifically, second-order functions take as input an *expression* closure, a *guard* closure, and an argument representing a list of values. For each value in the list, the functions apply the guard closure to that value and evaluate it. If the guard closure evaluates to True for a particular value from the list, the functions apply the expression closure to that value and evaluate it. The functions then return a value computed from the results of evaluating the expression closures. The various second-order functions differ only in how the return value is computed from the results of the expression closure evaluations. The first argument to all the second-order functions is a list specifier that selects the values to which the closures are applied. The list specifier can be one of $SMLIST, EVENTLIST, STATELIST$, or $FAULTLIST$, causing a complete list of either the state machines, the events, the states, or the fault names (respectively) to be used. Alternately, the list specifier can be an explicit list of names in the form $\{\text{Name}_1, \text{Name}_2, \ldots\}$, in which case the specified names are used. The use of these explicit names must be made appropriately, i.e., if the names are state machine names, then they may be used only in places where state machine names are appropriate. Since the second-order functions are the only means to bind free variables, closures may be used only as arguments to these functions. The complete list of second-order functions is as follows.

for_all(ListSpecifier, Guard, Expr) This second-order function applies the guard closure

to all members of the list specified by ListSpecifier. The guard closure acts like a filter, and prevents further processing of list elements that do not satisfy the guard expression. The Expr closure is applied to those members of the list for which the Guard closure has returned True. The for_all function returns True only if the Expr closure evaluates to True for all the list members to which it is applied. For example, the predicate function:

$\text{for\_all}(SMLIST, [x, \neg\text{chk\_state}(x, \text{CRASH})], [x, \text{chk\_state}(x, \text{Elect})])$

returns True only if all the state machines that have not yet crashed are in the Elect state. If no members of the list pass the guard closure filter, then the function returns True.

if_any(ListSpecifier, Guard, Expr) This function returns True if there exists at least one member that passes the guard filter, and the Expr closure returns True when applied to it. If no members pass the guard filter, then the function returns False. Note that the syntax and semantics of the ListSpecifier and Guard arguments are identical to those in the for_all function. For example, the predicate function:

$\text{if\_any}(\{\text{Node}_1, \text{Node}_2, \text{Node}_3\}, [x, \neg\text{chk\_state}(x, \text{Coordinator})], [x, \text{chk\_event}(x, \text{Elect})])$

returns True if an Elect event occurs on a node in the specified list, even though the node is not the co-ordinator.

how_many(ListSpecifier, Guard, Expr) With the same syntax and semantics for the ListSpecifier and Guard arguments as for the for_all and if_any functions, this function returns the number of list members to which the Expr closure, when applied, evaluates to True. For example, the predicate function:

$\text{how\_many}(SMLIST, [x, 1], [x, \text{chk\_state}(x, \text{Coordinator})])! = 1$

returns True when the system has either fewer than or more than one coordinator. Note the use of the closure $[x, 1]$, which always returns True no matter what argument it is applied to.

The facility to nest closures in one another allows the nested use of the second-order functions. For example, the predicate:

$\text{if\_any}(STATELIST, [x, 1], [x, \text{for\_all}(SMLIST, [y, \neg\text{chk\_state}(y, \text{CRASH})], [y, \text{chk\_state}(y, x)])])$

returns True if all the nodes in the system that have not crashed are in the same local state. However, it must be noted that to achieve the nesting effect, the names of the free variables (x, y) in the nested closure must be unique. Otherwise, the free variable in an inner scope overrides the free variable in an outer scope. In the implementation of the measures language, closures are supported through maintenance of a runtime environment for each closure; the

environment is implemented by a hash table that maintains the mapping between variable names and their values.

2.2.4 Incorrect Injection Removal

Section 2.2.1 described the algorithm used to identify whether a fault has been injected correctly, incorrectly, or not at all. This identification is dependent only on the fault trigger, and hence is done for all experiments. However, the action to take when a fault has been injected correctly or not at all is specific to the type of experiments being conducted and sometimes even to the measures being computed. For example, if a system is being evaluated for sensitivity to crash failures during error recovery, then the experiments in which the crash failure was not injected in the error recovery states must be discarded. However, those "faults" that trigger workload generation in the system at the beginning of each experiment may not be so sensitive to injection in the correct state, and incorrect injection of these faults may be ignored. Moreover, it frequently happens that the designer of the experiment did not even intend for all the defined faults to be injected in each experiment. An example is the scenario in which a fault is to be injected in the coordinator of a distributed protocol. Since the coordinator is not known before run-time, triggers must be defined on all nodes. However, only one trigger fires in every experiment.

A possible solution may be to allow tagging of each fault type with rules such as "if type = INCORRECT then ignore" or "if type = NO_INJECTION then discard" and have the analysis phase use these rules to discard experiments appropriately. However, it is observed that the actions to take on incorrect or no injections may even depend on the measure being evaluated. For example, when measuring the sojourn time of a state when faults are injected into it, it may be important to discard experiments in which faults were not injected correctly in that state. However, when measuring the effect of the faults on a much later state, it may not be important that the faults were injected precisely; the fact that they were injected may be enough.

These observations have led to the inclusion of support for processing incorrect injections or no injections within the measures language itself. Two functions have been added to the language to provide this support. They are as follows.

chk_fault(StateMachine, Fault, Label) This function returns a Boolean value if the given fault on the given state machine was labeled with the given label by the incorrect fault identification algorithm. Here, the label can be $INJECTION$, $NO\_INJECTION$, $CORRECT$, or $INCORRECT$.

chk_fault({SM1, SM2, ...}, Fault, Label) This function returns the number of state machines from the given list on which the given fault was labeled with the given label by the incorrect fault identification algorithm.

These functions can be used in the subset filter, the predicate function, or the observation function. When used in the subset filter, the check fault predicates can be used to prevent experiments from being used for the measure based on the labels on each of the fault injections. For example, a subset selection filter of:

$\text{chk\_fault}(\{\text{Node}_1, \text{Node}_2, \text{Node}_3\}, \text{Fault}_1, CORRECT) == 1$

can be used to keep only those experiments in which the fault `Fault1` was correctly injected on exactly one node in the system. When used in the predicate function, the check fault predicates can extend the power of the measures language by allowing measures to incorporate information about faults into the generation of predicate timelines. For example, the predicate function:

$\text{if\_any}(SMLIST, [x, \text{chk\_fault}(x, \text{Crash}, CORRECT)], [x, \text{chk\_state}(x, \text{Recovery})])$

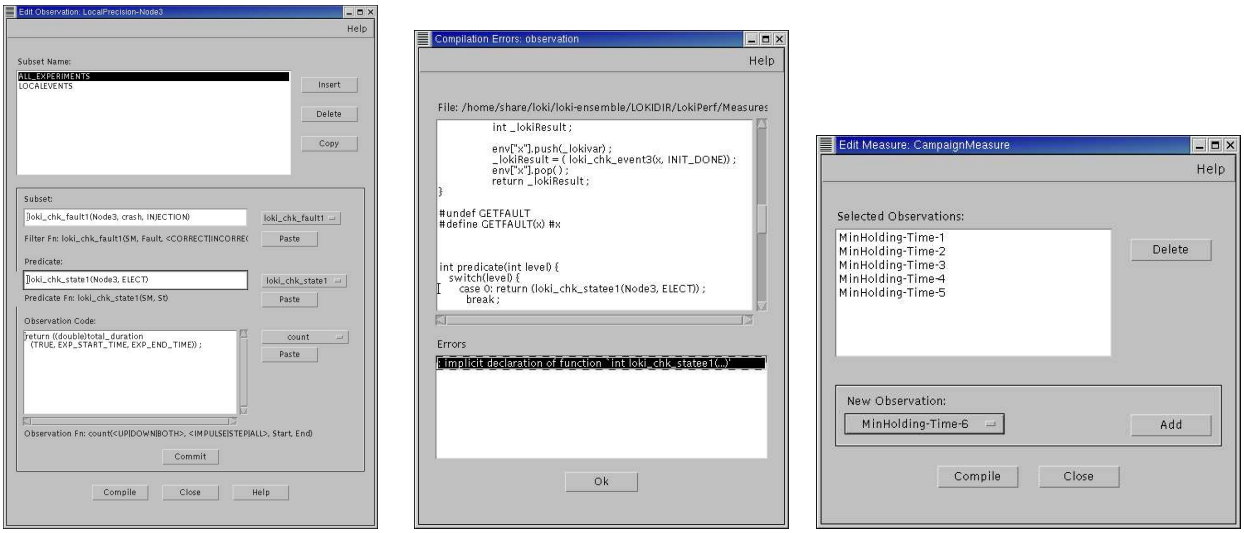
can be used to generate a timeline that is true only whenever a node on which a crash failure was correctly injected is recovering from it. This support for fault information can be extremely useful, especially during computation of conditional measures.

2.3 Measures Implementation

The implementation of the Loki measures language as described in Section 2.1.3 was performed as part of our work. This section gives an account of the implementation and techniques used to compute the measures described by the measures language. The measures implementation consists of two parts: the front end and the back end. The front end provides the measures user with a GUI to help define measures. When the measures are defined in the front end, it parses them, processes any closures present inside the measures, and generates C source code and Makefiles that implement the measures. The C source code implements measures using uses generic measure computation functions defined in the back end. After the front end has generated the measure source code, it also provides the user with facilities to compile the code, execute it and view the results.

2.3.1 Measures Front End

A substantial portion of the measures front end consists of a Graphical User Interface (GUI) written in Java. The front end contains editors to specify and edit both study-level and



(a) Study-Level Measure Editor (b) Error Dialog (c) Simple Sampling Measure Editor

Figure 2.5: Loki Measure Definition Editor

campaign-level measures. Both the editors are invoked from the Loki campaign manager (the main Loki window). When the study-level measure manager is invoked, the user is shown a list of the study-level measures defined for the current campaign. The user can create, delete, or copy measures using the GUI. If the user chooses to edit a measure, then he/she is presented with the study-level measure editor, which is shown in Figure 2.5(a). The editor shows the tuples defined for the current measure as seen in the figure. The user can add, delete, copy, or edit tuples. Each tuple is named in order to implement the named tuple extension described in Section 2.2.2. When a tuple is selected for editing, the subset selection function, the predicate function, and the observation function can be entered or edited. The measure editor provides drop-down boxes that list the available predicates, functions, and previous tuple names that can be used in each of the definitions. Context-sensitive help that shows the function syntax is also provided.

When the user saves a study-level measure definition, the front end writes a specification file ('.spec' extension) with the definition in it. Although the main algorithm for computing the timelines and the definitions of the standard measure functions, such as the chk_state, chk_event, and second-order functions, are in the back end, C code must still be written specifically for each measure. The code contains the definitions for the subset selection filter and the predicate and observation functions. The syntax used by these functions conforms to standard C syntax, and hence the definitions entered by the user can be used in the C code file directly, with one exception. C does not provide any support for closures or any other higher-order function mechanisms. Hence, the predicate function must be parsed for

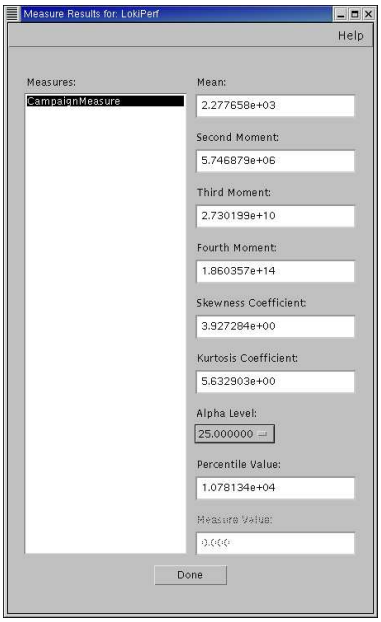


Figure 2.6: Measure Computation Results

closures. When closures are found, they are written in the form of separate C functions, and are passed around through the use of C function pointers. Support for nested closures is achieved by having a global environment. The environment is a hash-table-based mapping of symbol names to stacks of symbol values associated with each name (a stack is needed because a nested closure might contain a free variable with the same name as one in an outer closure). The top value on the stack represents the symbol value that is currently in scope. The environment is used for looking up the values of all free variables in a closure. An "observation array" is also defined as part of the generated code file. It is used to store observation values as they are generated by each tuple. Then, any subsequent references to the observations used in previous tuples (the named tuple extension) can be satisfied from the array. Finally, the study-level measure editor also generates a makefile that can be used to compile the measure code and to compute the values of the measure by executing its code. When the user chooses to compile the code, the measure editor parses the output of the compiler, and provides an error-checking GUI that shows the user where errors (if any) occurred in the code. This error-checking GUI is shown in Figure 2.5(b).

The campaign-level measure manager is similar to the study-level measure manager and also allows the user to create, delete, and copy campaign-level measures. A campaign-level measure is edited using the campaign-level measure editor. If the campaign-level measure is a simple sampling or a stratified weighted type of measure, then the user is presented with a list of study-level measures defined on studies in the campaign. He/she must select

those that are to be combined to form the campaign-level measure. The simple sampling measure editor is shown in Figure 2.5(c). In the case of stratified weighted measures, the user is also required to assign weights to each of the selected study-level measures. If the campaign measure is of the stratified user type, then the user is presented with a input box in which he/she must write code that computes the measure as a function of the averages of selected study-level measures. When the campaign-level measures are saved, the measures front end generates C code to combine the study-level measures to obtain the measure values. Statistical functions provided by the back end are used to compute the statistical properties of the computed measure values. Finally, a makefile is also generated to compile the measure and execute it. The same error-checking GUI that was used for study-level measures is used to perform error checking during compilation. When the measure is executed through the front end, the front end runs the measure, parses the output, and displays the statistical properties of the measure in a dialog box, as shown in Figure 2.6.

2.3.2 Measures Back End

The measures back end contains the actual algorithms to compute the measures, as well as the functions that are used by the measures code generated by the front end. The back end is a library that is compiled along with the code files generated by the front end to obtain the measure executable. Algorithm 2 shows the algorithm that is used to compute a study level measure. The algorithm processes all of the experiments in a study one by one. When a new experiment is loaded, the first thing that is processed is its global timeline. Recall that the events in the global timeline occur over intervals of time rather than at precise instants. This poses a problem because accurate estimation of the kind of constructs used by the measures language (such as computing the order between transitions, or computing sojourn times) is not possible if events are defined over intervals rather than instants (and no other information is available about the distribution of the exact event occurrence time within each interval). To solve this problem, Algorithm 2 first averages the intervals to get a single time instant for each event occurrence. Although this is an approximation and can cause errors in the final measure estimation, the error is not likely to be large. The reason is that in a typical LAN environment (for which Loki is designed), the distributions for network latency in one direction are very similar to the distributions for network latency in the opposite direction (LANs typically have symmetric channels). Hence, the likelihood that events will occur near the centers of their time intervals is high. The averaging process yields a timeline on which events occur at single instants of time, thus allowing the processing of tuples in the measure.

Algorithm 2: Algorithm to Compute a Study-Level Measure
Input:
$E = \{E_i | i = 1, \ldots, |E|\}$, an ordered set of experiments in the study,
$G = \{G_i | i = 1, \ldots, |E|\}$, an ordered set of global timelines, one for each experiment, where each G_i is a sequence of tuples $(node, event, t_{low}, t_{high})$,
$M = \{Tuple_i | i = 1, \ldots, |M|\}$, a study-level measure where each $Tuple_i$ is of the form (FilterFn$_i$,PredicateFn$_i$,ObservationFn$_i$),
$N = \{N_i | i = 1, \ldots, |N|\}$, an ordered set of nodes used in the study.
Output: The value of the study-level measure M for each experiment in E

```
ComputeStudyLevelMeasure(E,G,M,N)
(1)    for e ← 1 to |E| /* For each experiment */
(2)      for i ← 1 to |G_e|
(3)        AvgTime_i ← Average event occurrence time for event (G_e)_i
(4)      sort(G_e) with AvgTime_i as key
(5)      for t ← 1 to |M| /* For each tuple */
(6)        EvalInstants ← Explicitly used time instants from PredicateFn_t
(7)        sort(EvalInstants)
(8)        i, j, k ← 1, time ← 0, PredValue_0 ← False
(9)        globalstate ← {InitialState_1, ..., InitialState_|N|}
(10)       while i ≤ |G_e| ∨ j ≤ |EvalInstants|
(11)         if j > |EvalInstants| ∨ AvgTime_i ≤ EvalInstants_j
(12)           (N_p, event, t_low, t_high) ← (G_e)_i
(13)           globalstate_p ←NextState(N_p, event)
(14)           time ← AvgTime_i
(15)           i ← i + 1
(16)         else
(17)           time ← EvalInstants_j
(18)           j ← j + 1
(19)         if PredicateFn_t(globalstate, time)≠ PredValue
(20)           PredTimeline_k ← time
(21)           PredValue ← ¬PredValue
(22)           k ← k + 1
(23)           /* Check if change in predicate timeline is an impulse */
(24)           if PredicateFn_t(globalstate, time+(Δt)_min)≠ PredValue
(25)             PredTimeline_k ← time
(26)             PredValue ← ¬PredValue
(27)             k ← k + 1
(28)       Observations_t ←ObservationFn_t(PredTimeline)
(29)       if ¬FilterFn_t()
(30)         break/* Discard Experiment */
(31)       if t = |M| /* Print the final observation value */
(32)         print Observations_t
```

After the averaging process is completed, the experiment being processed is made to go through each tuple in order, until the experiment either is filtered out by some tuple, or passes through the final tuple. In the latter case, the final observation value is written out as the value of the study-level measure for that experiment. The first step during the processing of a tuple is the computation of a predicate timeline. The predicate timeline is assumed to start with a value of false. The back end computes the times at which the truth value of the timeline changes. To do so, it scans the global timeline in ascending order of time. The global state of the system is updated at every event occurrence. The predicate function is evaluated at the time of every event occurrence and at every time explicitly specified in the predicate function to check if it changes its truth value. Additionally, the predicate function is also evaluated at the smallest time increment after each event occurrence to check for impulses on the predicate timeline. The work of querying the timeline is done by the chk_state and chk_event functions that are defined in the back end. Since the predicate timeline can change only if the global state changes (at local event occurrences) or at times that are explicitly specified in the predicate function, computing the global state only at these points is sufficient to generate the predicate timeline. The predicate timeline is represented as an array of times at which the truth value of the timeline changes. Next, the user-defined observation function is executed to compute the observation value from the generated predicate timeline. The functions count, outcome, duration, instant, and total_duration that query the predicate timeline [Cha00] are also implemented in the back end. The chk_fault functions that query the fault injection labels are defined in the back end as well. They parse and use the output generated by the incorrect fault identification algorithm that runs in the analysis phase. Finally, the back end executes the filter function for the tuple in order to determine whether the next tuple should be processed for the current experiment, or whether the experiment should be discarded. The filter function can use the current or previous tuples' observation values. If the current tuple is the final tuple, and the experiment passes the filter, then the final observation value is written to the output file. The complete API available to the measures user can be found in Appendix A.

2.3.3 Sojourn Time Computation Tool

The Loki measures implementation also includes a tool called `lokihtime` that computes the sojourn times for all the states of all the nodes in an experiment. Like the measures back end, this tool also parses the global timeline for the experiment and averages the event occurrence intervals. After doing so, the tool then scans the global timeline and computes the sojourn time for each state in each state machine and the number of times each state was entered.

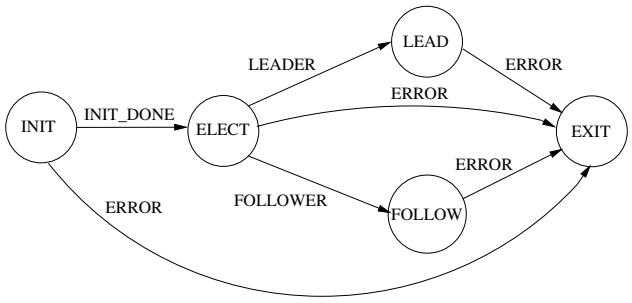


Figure 2.7: State Machine Used for Loki Performance Evaluation

All the experiments in a study can be processed in a single batch. The tool has four modes of output. The `experiments` mode outputs the sojourn time data for each state of each state machine for each experiment in a study. The `expoverall` mode computes the distribution of the sojourn times for each state across all the nodes in the system (this is useful when all the nodes follow the same state machine) and prints the average and standard deviations of the sojourn time for each state in each experiment. The `summary` mode computes the distribution of sojourn times for each state in each state machine across all the experiments in the study. The means and standard deviations of these distributions are printed for each state in each state machine. Finally, the `overall` mode computes the distribution of sojourn times for each state across all the state machines and all the experiments and prints the mean and standard deviations for this data. Again, the `overall` mode is useful only when all the nodes in the system have identical state machines. The sojourn time computation tool is very useful for quickly analyzing the overall behavior of a system without requiring computation of any measures. It can also be used along with the state machine descriptions for generating semi-Markov Chain models for the nodes in the distributed system.

2.4 Performance Characteristics

This section presents metrics for the Loki implementation that quantify the precision, intrusiveness, and granularity of the fault injection mechanism and the accuracy of the analysis. These metrics illustrate the capabilities and limitations of the implementation and put into perspective the fault injection results presented in Chapter 4. The results in this section were obtained using three 1GHz Pentium III-based hosts with 256MB of RAM and a 100Mbps switched Ethernet network. The application used is the implementation of a simple leader election protocol with three nodes, one on each host. The nodes are specified by identical state machines. The state machine specification is shown in Figure 2.7. Nodes start in the

`Init` state of the state machine in which they establish connections with the other nodes. Then, each node transitions to the `Elect` state in which the leader election is carried out. Each node sleeps for 20ms in the `Elect` state, but this time can be varied to control the holding time for this state. Depending on the result of the election, nodes then transition to the `Leader` or `Follower` state, after which the experiment ends. Fault triggers are set to inject a crash failure into a single node when all three nodes are in the `Elect` state.

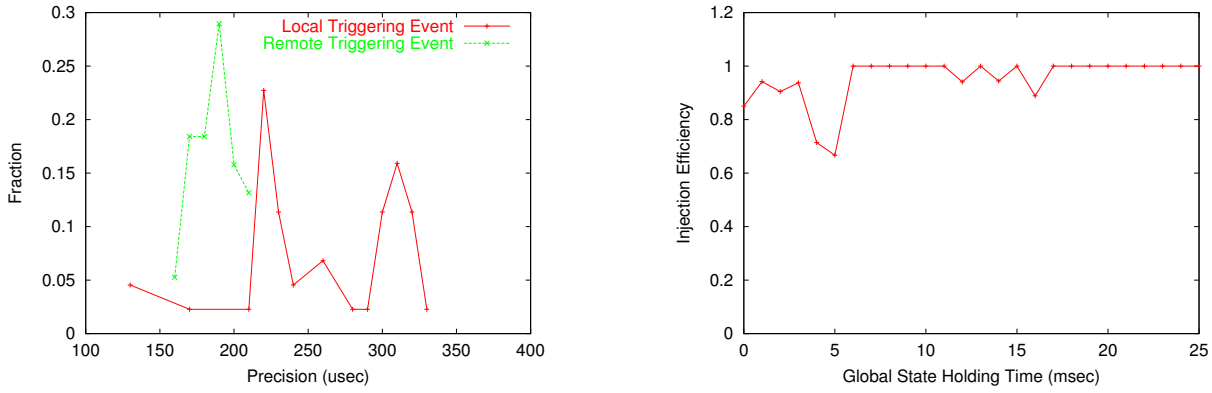
2.4.1 Fault Injection Mechanism Metrics

These metrics quantify the performance of the fault injection mechanism in the Loki runtime. For the purposes of these metrics, the local event that causes a fault trigger to become true is called the *entry event* for that trigger, while the local event that causes the trigger to become false is called the *exit event*. The global state in which the user intends to inject the fault is called the *trigger state*.

Imprecision

The imprecision of the fault injection mechanism is defined as the amount of time it takes for a fault to be injected into the system after its trigger has become true, i.e., the time difference between the entry event and the injection event. The imprecision measurements can be divided based on whether the entry event and the injection event occur on the same host, or on different hosts. When the two events occur on different hosts, then a direct comparison of occurrence times cannot be made because the events are represented by intervals on the global timelines. We average the intervals to estimate the occurrence times. Since the LAN is a symmetric medium, averaging is expected to yield fairly accurate results due to reasons described in Section 2.3.2.

Figure 2.8(a) shows the distribution of imprecision for both local and remote events. These results were obtained by conducting 150 experiments, and the actual values of precision were collected into bins of 10 μsec. From the figure, it is seen that the maximum imprecision observed was about 350 μsec. Hence, it is reasonable to expect Loki to inject faults correctly into states that have holding times as low as 500 μsec. Also, it can be seen that the imprecision when the fault was triggered by a remote event is actually lower than when the fault was triggered by a local event; this is counter intuitive. However, it must be noted that the timeslice of the operating system used here was 10msec. This means that even slight differences in the exact scheduling of the application and Loki runtime threads for the local and remote triggering event cases could easily have led to this difference.



(a) Imprecision (b) Injection Efficiency

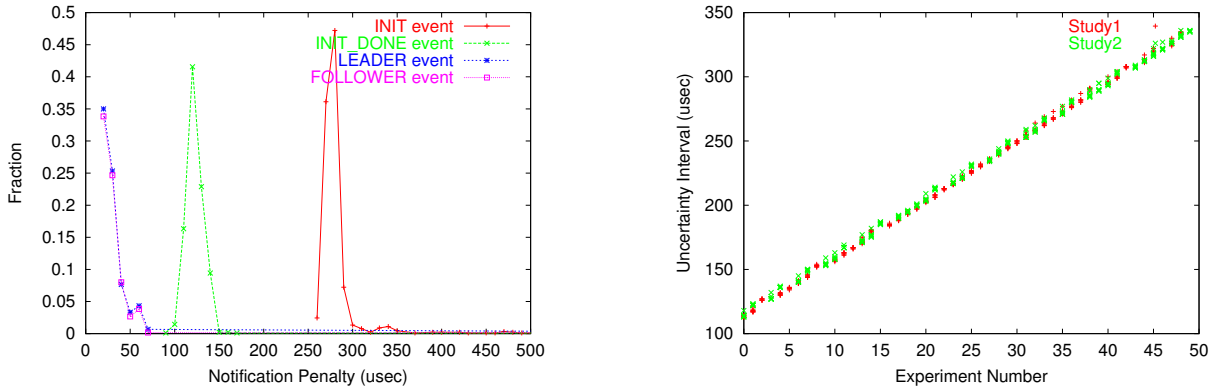
Figure 2.8: Imprecision and Efficiency of Fault Injection

Injection Efficiency

The injection efficiency of the fault injection mechanism is defined as the fraction of faults correctly injected into a trigger state with a certain holding time. As the holding time of the trigger state decreases, the injection efficiency eventually falls, making it harder to inject faults. Hence, injection efficiency is important because it ultimately limits the granularity of fault injections that may be performed. Injection efficiency depends on several factors, including the imprecision of fault injection, trigger state holding time, and the event uncertainty intervals. A detailed discussion of how these factors affect injection efficiency can be found in [CLJ+02]. Figure 2.8(b) shows the injection efficiency plotted against a varying trigger state holding time. We varied the holding time by inducing an artificial delay into the `ELECT` state of the leader election protocol, and conducted 800 experiments. The resulting holding times were collected into bins of 1 msec each. As seen in the plot, the injection efficiency for trigger state holding times of 1 msec or higher are quite high. This is expected, because the imprecision of the fault injection is limited to 350 μsec, as shown earlier. The efficiency becomes perfect for holding times higher than 20 msec, which corresponds to two timeslices of the operating system.

Notification Delay

The notification delay is defined as the wall-clock time taken to notify the Loki runtime of a local event. Since this event notification is done by instrumenting the application, the notification delay is a source of intrusion. Hence, it is desirable for this delay to be as low as possible. Figure 2.9(a) shows the distribution of notification delays for the events in the leader election application. To obtain these results, we conducted 200 experiments.



(a) Notification Delay

(b) Variation in Uncertainty Interval

Figure 2.9: Notification Delay and Uncertainty Intervals

In the experiments, a dummy fault was injected when one of the nodes was elected as the leader (i.e., on the `LEADER` event). This was done so that we could observe whether an event notification triggering a fault would have any effect on the notification delay. The `INIT_DONE` event is the only event for which notifications were sent to all the other nodes. The `INIT` event was the first event generated in every experiment, while the `FOLLOWER` event is a simple event for which no notifications were generated. The `FOLLOWER` event thus represents the base case for comparison. The resulting notification delays were categorized into bins of 10 μsec for inclusion in the plot.

It can be seen from the figure that the notification delay distributions for all four events have well-defined modes and are quite narrow. However, all of them have long tails. These tails have very low probabilities of occurring, and have been truncated in the figure. The reason for the long tails is that the operating system timeslice is 10 msec. If there happens to be a context switch during the notification, then the wall-clock time for performing the notification increases substantially. Another interesting feature of the plot is that some distributions have different modes. The `INIT` event has the largest mode, because it is the first event to be generated. When the Loki runtime receives the first event, it initializes several data structures used for tracking the node. This results in the higher notification delay. Since the `FOLLOWER` event was the baseline event, it has the lowest mode, as expected. However, the `LEADER` event also has an almost identical distribution, indicating that the fact that it was a fault triggering event had no effect on its notification delay. Finally, the `INIT_DONE` event had a higher notification delay that the base case, because all the other nodes were notified of its occurrence. Finally, the most important thing to observe from Figure 2.9(a) is that all the modes (except for the initial event) are less than 150 μsec indicating a fairly low notification overhead for most applications.

2.4.2 Analysis Accuracy Metrics

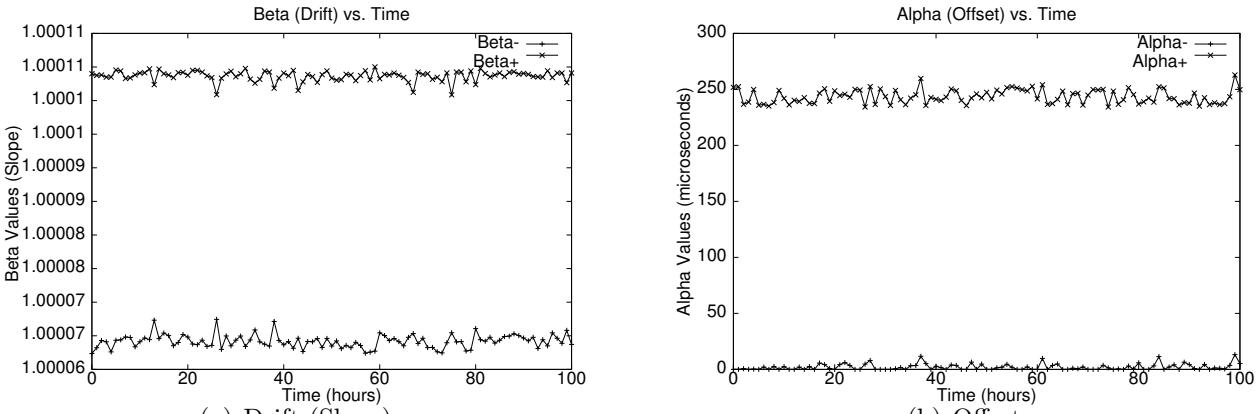
The following metrics quantify the accuracy of the global timelines generated in the analysis.

Uncertainty Interval Variation

As defined in 2.2.1, uncertainty intervals are caused by the finite resolution of the clock synchronization algorithm. The lengths of the intervals are important, because they affect how events are ordered, and the injection efficiency. If they are large, they can also affect the values of the measures. Figure 2.9(b) shows the lengths of these intervals on the testbed used in our study, and how this length varies over several experiments in a single study. We obtained the graph by conducting two studies with 50 experiments each. Each experiment was less than a minute long, and the time between consecutive experiments was 15 seconds. It can be seen that for both studies, the uncertainty intervals begin at a little over 100 μsec, but increase linearly at the same rate as the study progresses. This phenomenon is caused by the nature of the clock synchronization algorithm used by Loki. The algorithm assumes that the hardware clocks of the hosts on which experiments are run have a linear relationship between them. It tries to estimate upper and lower bounds on the slope (drift) and offset for the relationship. These bounds are used to compute the global timeline. The uncertainty intervals result from the fact that these bounds differ by small amounts. As the study progresses, these small differences cause the uncertainty intervals to grow larger. However, it can be proved that there exists an upper bound on the length of the intervals that is independent of the length of a study. This upper bound is a constant factor of the maximum single trip network delay [CLJ+02]. To slow down the increase in uncertainty intervals, an enhanced version of the clock synchronization algorithm [Hen98] can be used if necessary.

Constancy of Clock Drift and Offset

The off-line clock synchronization algorithm used by Loki relies on the assumption of a linear relation between the hardware clocks of hosts in the testbed [Hen98]. It has been suggested in the literature that this linearity assumption holds [EK73] for reasonable periods of time. To experimentally verify this claim, and ensure that the lengths of experiments conducted in our study fall within the reasonable limits for linearity of clocks, the drifts and offsets between two machines were computed using the clock synchronization algorithm over a period of several days (which was longer than any experiments conducted in this study). The results of the experiments are shown in Figure 2.10. The graphs show the upper and lower bounds for the drift and offset between the two machines computed every hour over a period of 100



(a) Drift (Slope) (b) Offset

Figure 2.10: Variation of Drift and Slope over Time

hours. The figure shows that drifts and offsets did not vary appreciably over this period, and suggests that the linear clock assumption would hold for other experiments as well.

2.4.3 Comments on Performance Metrics

From the various performance metrics, it can be concluded that on the given experimental setup, results obtained through Loki can be relied on for resolutions and precision of around 500 μsec. Also, the experiments reveal how the length of the uncertainty intervals varies as a study progresses. Our measurements should help users of Loki understand the tool's capabilities and limitations. For fault injection studies that require relatively low resolution and precision, our measurements should be sufficient. However, for very critical and finely granular experiments, the measurements should be retaken on the actual testbed to be used, in order to ensure validity of the results. It is important to note that the precision and the notification delay are artifacts of the Loki implementation. Precision could be improved by speeding up the parsing of fault triggers, while notification delay can be improved (or eliminated) by using non-intrusive notification techniques such as external observation of the program output. However, the length of the uncertainty intervals is lower-bounded by the latency of the network used, and hence possesses a non-zero lower bound. The injection efficiency is a function of both precision and the uncertainty interval, and therefore has some room for improvement.

Chapter 3

Ensemble Study Specification

This chapter describes the specifications for the evaluation of the Ensemble group membership protocol. It begins by describing the role of a group membership protocol in group communication, and how that has dictated the kinds of experiments performed and the faults to be injected in this case study. That explanation is followed by a specification of the fault models used, and the environmental conditions set up for the study. Then, the state-machine model of the membership protocol and the technique used to derive it are presented. Finally, we give specifications of the fault triggers used and the desired measures. The specifications are in terms of the state-machine model of the system.

3.1 Evaluating a Group Membership Protocol

To evaluate a group membership protocol, it is important to understand its role in the operation of a group. Figure 3.1 shows a global-state diagram that describes this role. In

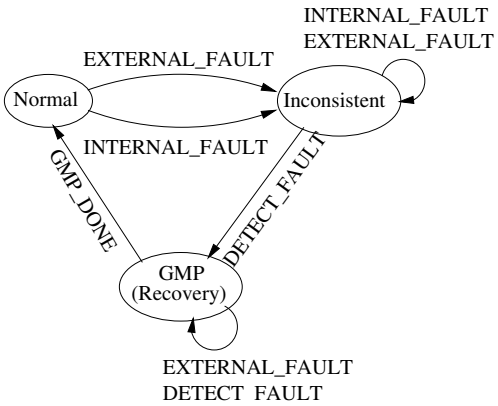


Figure 3.1: Group Membership Protocol as a Fault Tolerance Mechanism

the normal operation of a group, all nodes have a consistent view of the group membership. This is a safe state of the distributed system, and is represented by the `Normal` state of the figure. Any event that causes a change in group membership causes a loss of consistency, since the nodes that have entered or left the group do not have the same view of the group membership as the other nodes. Thus, membership change events can be said to cause the distributed system to go into an unsafe state, and are hence considered faults. The group membership protocol can then be seen as a mechanism to restore the safe state of the system; hence, it is a fault tolerance mechanism. In the figure, these membership change "faults" are categorized as *Internal Faults* and *External Faults*. Internal faults are those events that are generated from within the application that drives the Ensemble stack (group leave events), or from some part of the stack itself (partition merge events). External events are events over which the the stack has no control (such as crash failures or network partitions).

Based on the role of a group membership protocol described by Figure 3.1, there are three measures that characterize its behavior. The first is the coverage (i.e., the probability that the membership mechanism will detect a membership "fault"), which is defined as the probability that the state machine in Figure 3.1 will transition from state `Inconsistent` to state `GMP` in response to an internal or external fault. The second measure is the time taken to detect an inconsistent group membership before the membership protocol is invoked (i.e., the time spent in the `Inconsistent` state). This time is called the *detection time*. Finally, the third measure is the time taken by the membership protocol to recover from the membership change. This is the time spent in the `GMP` state of the state machine. It is also the time for which the membership protocol blocks message traffic, and is called the *blocking time*. Ensemble employs a mechanism based on heartbeat messages and timeouts to detect crash failures and network partitions. This kind of failure detector is *complete* [CT96] because it eventually detects all failures. Internal faults are always "detected" since they are generated by the stack itself. Hence, for our case study, coverage is assumed to be 100%. The detection time for the crash failures and partition merges is strongly dependant on the timeout interval, which is a tunable parameter. Hence, our case study focusses on the blocking time, which is not so easily controllable by the user, and which is one of the primary causes of the unavailability caused due to group membership.

3.2 Fault Model

This section categorizes the kinds of internal and external faults that cause the invocation of the group membership protocol, and describes the fault models used in the study.

3.2.1 Basic Faults

The group membership protocol handles several group membership events. They include node joins, voluntary node leaves, forced eviction of a node due to suspicious activity, node crashes, network partitions, and network partition merges. Hence, a complete evaluation would have to consider all these events separately. However, a close examination of the protocols reveals that these events can be grouped into two categories such that the protocol behaves similarly for all events in the same category.

The first category is the *node entry* category. This category includes the partition merge and node join events. The protocol implements node joins by first creating a singleton partition that consists only of the new node. This singleton partition then merges with the group using the regular partition merge protocol. The second category is the *node exit* category. It includes node crashes, voluntary node leaves, node evictions, and network partitions. These events differ only in the way they are detected. A crashed node is detected by a heartbeat-timeout mechanism. A network partition is detected the same way, but multiple nodes may be suspected. Those two types of node exits involve timeout intervals. On the other hand, a voluntary node exit is signalled by the node that wishes to leave. Hence, detection is instantaneous. Forcible eviction of a node due to suspicious behavior is done as soon as a sufficient number of members suspect the node.

Due to the similar ways these events are handled, the fault model for this study consists of one representative event from each category. A node join event is chosen from the node entry category, and a node crash event is chosen from the node exit category. These two fault models are implemented as follows:

Crash Failures are implemented via an induced segmentation fault. The fault injection routine dereferences a *null* pointer, thus causing the operating system to send a signal to the application and crash it. However, because the fault injection routine and the Ensemble application execute in different threads, the Ensemble application thread continues to run and execute the membership protocol until the process receives the OS signal. It may take as long as an entire scheduler time-slice for the process to receive the signal. To prevent the delay from affecting the precision of the injection, we made a small modification to the Ensemble event scheduler to make it stop processing events when a particular flag is set. The fault injection routine sets this flag, thus causing the Ensemble stack to freeze until the signal arrives.

Node Joins are implemented when the fault injection routine is made to fork a new process that executes the secure shell (ssh) application. The ssh application starts a new

instance of the workload generator application on a specified remote host. The new instance of the workload generator application then automatically finds and joins with the main group, thus causing a node join.

3.2.2 Fault Combinations

We can make the fault model more representative by considering the fact that faults can continue to occur even while the group membership protocol is executing. Hence, two sets of experiments are conducted. In the first set of experiments, only one membership change event is injected into each run of the system, and the resulting behavior of the membership change protocol is studied. Those experiments are called the *single fault experiments*. They seek to evaluate the performance of the protocol under conditions of varying system parameters. In the second set of experiments, an additional membership change event is injected into the system while the membership protocol is handling the first one. The second set of experiments, which are called *correlated fault experiments*, allow an evaluation of the robustness of the membership change protocol to additional faults. The second injection is parameterized by the global state the system should be in at the time of the membership event injection. Since there are two types of membership change events (node entry and node exit), four combinations of correlated injections are possible. Indeed, since node exit events may happen due to circumstances beyond the protocol's control (such as a node crash or a network partition), they can occur while the protocol is handling another membership event. However, node entry events are always initiated by the Ensemble stack itself, and are blocked while the membership protocol is executing. Hence, only two combinations of correlated faults are possible viz. a node entry followed by a node exit, or a node exit followed by another node exit. From a practical viewpoint, correlated node exits are especially important, since events such as crash failures and network partitions frequently happen together in real life. Hence, only correlated node exits (a node crash followed by another node crash) are considered in this study. However, the behavior for a node entry followed by a node exit event can be studied using the same technique.

3.3 Experimentation Environment

The environment in which the group membership protocol executes is described by two primary parameters. They are the workload offered to the group communication system by the applications using it, and the size of the group. The following discussion describes how these parameters were chosen and varied.

3.3.1 Workload

One of the most important characteristics of an application that runs on a group communication system is the message workload it generates, which is the only characteristic considered. The message workload is specified in terms of the length and type (point-to-point or broadcast) of messages generated and their generation times. Ideally, the workload should be representative of reality. However, since Ensemble is a generic group communication system intended to be used in a wide variety of applications, it is very hard to argue that any one type of workload is representative. Hence, the application message generation process at a single node (the workload) was modelled by a Poisson process with parameter λ, which is varied in different studies. Due to their memoryless nature (i.e., knowledge of the arrival times of previous messages provides no additional information about the arrival times of future messages), Poisson processes are widely used models for unknown message generation processes. In the experiments, 100-byte-long fixed-sized broadcast messages were used. To generate the workload, we wrote an OCAML workload generator. It instantiates a new Ensemble stack and broadcasts a stream of messages with exponentially distributed (with parameter λ) random delays to the entire group. The application uses the compatibility application interface provided by the Ensemble distribution [Hay97b] and runs on a standard virtual synchrony stack with the sequencer-based total ordering protocol.

3.3.2 Group Size

The group size affects the group membership protocol because membership operations require that agreement operations be carried out between all members of the group. In order to ensure a fixed group size for each experiment, the workload generator application was designed to monitor the group size periodically at the start of each experiment. When a group of the required size is formed, then the application notifies the Loki runtime (via a local event notification) to begin monitoring the experiment and begins workload generation. After a single group membership operation is done, the application exits. This ensures that only one membership operation with the required group size is carried out in each experiment. In the subsequent text, the nodes in the group are named $\text{Node}_1, \text{Node}_2, \ldots, \text{Node}_n$; NodeFail; and NodeMerge. All the nodes (including NodeFail and NodeMerge) execute the same workload generator (with the same workload rate λ), and are monitored by identical state machines. The nodes NodeFail and NodeMerge are named differently since they are the ones on which crash failures are injected and new nodes are instantiated, respectively.

3.4 Techniques for State Machine Generation

Existing specifications of the Ensemble stacks in the literature include 1) verbal descriptions of individual layers [Hay97a] without reference to inter-layer interactions and 2) a formal specification of some aspects of protocol behavior intended to prove correctness of the virtually synchronous behavior [HLvR99]. However, temporal specifications of the end-to-end behavior of the entire group membership protocol that take into account the interactions amongst the layers do not exist in the literature. Construction of such a specification is challenging for the reasons described below.

3.4.1 Challenges

An instantiation of the Ensemble communication stack is a highly layered entity composed of many (typically 15-20) micro-protocol layers. The layers communicate with each other using the event-based model described in Section 1.3. Each layer processes only one event at a time, but the system may schedule independent events in multiple layers concurrently. Thus, at a given instant of time, there may exist several "threads of control" in the stack. These concurrent threads may be partially ordered in several ways. The existence of concurrent threads of execution and the lack of a single execution order amongst threads greatly complicate the process of abstracting the temporal operation of the stack with a single state machine.

A naive approach is to abstract each layer with a state machine in isolation and then compute the product to get a state machine for the entire stack. However, this approach cannot be used for Ensemble, because the large number of layers would lead to an explosion in the number of states, a large portion of which might not even occur in practice. Additionally, the presence of partial order amongst concurrent threads also exacerbates the state space explosion problem. The state space explosion could be limited through exploitation of the properties of the protocol behavior. However, such automatic reduction is a very heavy weight approach whose power is not needed for this case study.

To tackle the state-space explosion problem, rather than consider individual layers in isolation, we treated individual membership events, such as crash failure, in isolation. Each membership event forms a distinct input to the membership protocol, and thus uniquely determines a set of paths through the state space. We then composed the individual state machines to form the state machine for the behavior of the membership protocol in response to multiple membership change events. The advantage of this approach is that the complexity of state machine composition is dependant on the number of distinct input events (of which

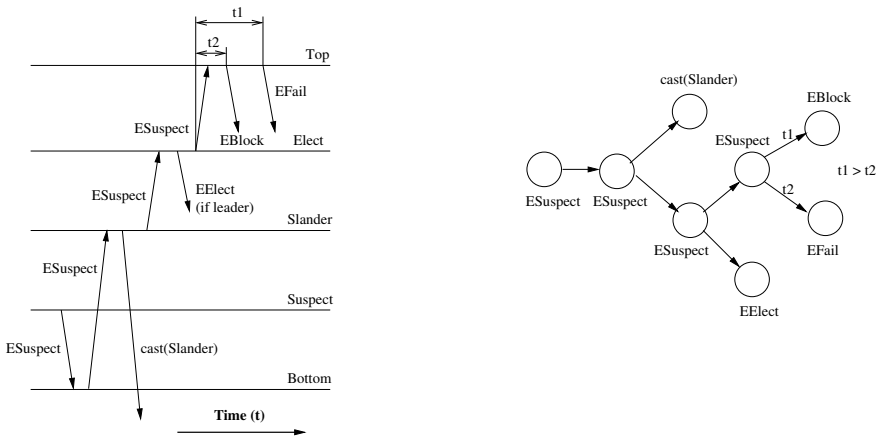


Figure 3.2: Event Chains

there are only few), rather than the number of layers. Moreover, the state machines for each individual event have several states in common. This leads to a substantial reduction in the state space. To deal with the problem of multiple threads of execution within the processing for a single event, a technique using "event chains" was used to construct the state machines for the individual events. This technique is described in the remainder of this section.

3.4.2 Event Chains

To generate the state machine for a specific membership event (e.g., a crash failure), we reverse engineered the source code for the Ensemble stack to construct event chains for the event. An *event chain* is defined as a directed graph in which the vertices represent inter-layer or external events and the edges represent direct causal relationships between the events within a single layer. The event chain is initiated by an "input" event. In the case of the Ensemble stack, the input event was the membership event for which the event chain was being constructed. In general, a causal relationship between events results in an acyclic graph, because multiple outgoing events may be generated during the processing of a single incoming event, and, conversely, an outgoing event at a layer may be generated as a result of waiting for several incoming events. Event chains effectively represent the temporal behavior of the inter-layer protocol that is composed of events bouncing back and forth within the stack.

A sample set of message exchanges in the stack and the associated event chain are shown in Figure 3.2. The complete diagrams of the message exchanges for the Ensemble group membership protocol are shown in Appendix C. The horizontal lines in the diagram on the left of Figure 3.2 represent the stack layers at which the events in this event chain are

processed. The vertical lines show the event flows, and thus are the vertices in the event chain, which is shown on the right. Time flows from left to right. Each edge in the event chain is assigned a weight, which is the time between the generation of the parent message and the generation of the child. The figure shows two such sample times, t_1 and t_2, along with the ordering relation between them ($t_1 > t_2$). In general, the exact values of these times are not known. However, in the case of the Ensemble stack, we often found it easy to deduce an ordering amongst the times based on knowledge of layer behavior and the FIFO event processing model. The weight of a path in the event chain is the sum of the weights assigned to all of the individual edges on that path.

3.4.3 Critical Path Computation

The key to obtaining a state machine description of the blocking behavior of the entire protocol is to compute the critical paths between the input event and the event that finally causes the protocol to unblock the group. The *critical path* between two vertices in an event chain is the path between them that has the greatest weight. The edges on this critical path represent possible states in a state machine model. An additional complication occurs whenever there is a decision procedure in a layer (e.g., is this node a leader or not?). In that situation, critical paths must be computed for each decision that can be made by the decision procedure and combined to form the single event state machine. Since the number of edges in the critical path for the Ensemble membership protocol was large, several adjacent edges of the same type were merged to form the states in the state model. The type of an edge (and thus its corresponding state) depends on the reason for the time delay associated with it. One possible reason is processing time within layers, which is represented by unshaded states in the state machine in Figure 3.3. Another possible reason is the time spent in application callbacks, which is represented by doubly circled states. Finally, time can be spent in waiting to receive an external event, such as a notification from another node; such states are shaded in the state machine diagram.

3.4.4 Combining Individual Event State Machines

After state machines were obtained for all the input events separately, we combined them to obtain a single composed state machine for the overall protocol behavior. In order to combine two state machines, states in the input state machines that have the same label in both the state machines, and are *outgoing transition indistinguishable* are merged to form a single state in the composed state machine. Two states are said to be outgoing

transition indistinguishable when any outgoing transitions with identical labels would have identical destination states in the composed state machine. If states with identical labels are not output transition indistinguishable, then their labels are changed to make them unique before they are added to the composed state machine. Transitions that go from a merged state to another merged state, and have the same transition label, are merged in the composed state machine. All other transitions are simply copied. If one of the membership events being combined is an event that can occur at any time during protocol execution, then the state machines for all the behaviors resulting from that event occurring in each state of the protocol must be combined.

We constructed two different state machines for the Ensemble stack. The first state machine was for the single fault experiments. Hence, it was formed by composing the state machines for the partition merge and "crash failure in the normal state" membership change events. The second state machine was for the correlated fault experiments. Hence, it was formed by composing the single fault state machine with the state machines for crash failures originating in each state of the single fault state machine. Although we performed the state machine composition manually for the Ensemble stack, the procedure is mechanical enough to be automated in the future.

3.5 State Machines

This section describes the state machines that model the operation of the Ensemble group membership protocol on a single node under both single and correlated fault injections. These state machines were obtained using the event chain technique described in Section 3.4.

3.5.1 Single-Fault State Machine

The Ensemble group membership protocol on each node can be decomposed into 5 phases, which are executed in succession. Figure 3.3 presents the single-fault state machine specification for the group membership protocol on an individual node. The state machine also contains additional states that are used for creating the environment in which faults are injected and measurement is done. These states are called *auxiliary states*. In the figure, solid transitions represent the local event notifications at each node that cause the state machine to change state. The dashed lines represent actual communication between the Ensemble stacks on different nodes (typically between the leader and followers) as the protocol executes. The state machines are not triggered directly by this communication, but

the communication may cause other operations in the stack which generate local events. The source and destination states for the dashed lines represent the states the sender and receiver are in when the communication occurs. States whose names start with the prefix `L` (or `F`) represent states that only the leader (or followers) of a group may enter. During the execution of the partition merge protocol, the partition that makes a merge request is called the *merging partition*, while the partition that responds to the merge request is called the *primary partition*. The corresponding leaders are called the *merging leader* and the *primary leader*, respectively. A brief description of each phase follows.

Auxiliary States

Auxiliary states and their associated transitions are used to track the execution of the protocol stack until the conditions are right for injecting faults into it. They are shown in Figure 3.3 in the boxes labeled "Auxiliary States." When the workload generator begins execution, the state machine begins in the `Init` state. The state machine remains in this state while the application forms an initial group of the required size. Hence, all local notifications except the group formation notification (`GROUP_FORMED`) are ignored in this state, as indicated by the `default` transition in the state machine.

When a group of the required size has been formed, the state machine transitions to the `Stabilize` state, where it waits for a certain number of messages to be generated by the node. This precaution is taken to remove any initial transients and to ensure that the results obtained are steady-state values. Once the requisite number of messages have been generated, the workload generator generates a `START_MEASUREMENT` notification that causes the state machine to transition to the `Normal` state. The evaluation then begins, and none of the state change notifications generated by the workload generator after this point are ignored. When all of the state machines have reached their `Normal` state, faults can be injected into the system, causing the group membership protocol to begin execution.

After the group membership protocol has finished execution, the state machine transitions into the `Done_GMP` state, shown in the bottom right-hand side of Figure 3.3. This transition indicates that the experiment is complete and that Loki should terminate the application. Further events are ignored by the state machine except the `EXIT` event, which occurs when the application quits, and causes the state machine to transition to the `EXIT` state. In this manner, we ensure that for every run, exactly one instance of the group membership protocol is evaluated under steady state conditions.

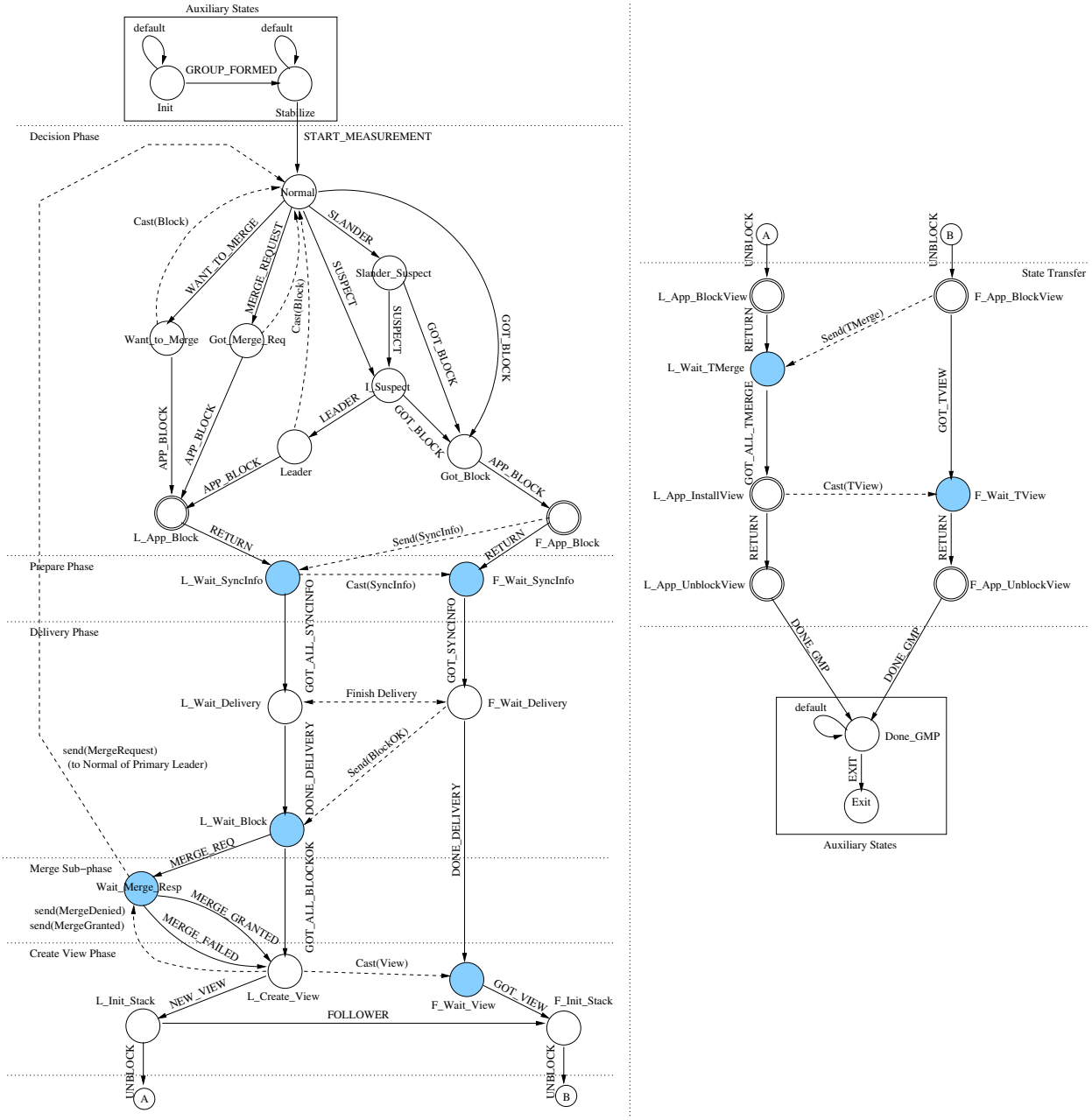


Figure 3.3: State Machine for the Single-Fault Injections

The Decision Phase

The first phase in the membership protocol is the *Decision Phase*. This phase corresponds to the "Inconsistent" state shown in Figure 3.1. In this phase, Ensemble enters the membership protocol because of some triggering event. The corresponding part of the state machine shown in Figure 3.3 models this situation for crash failures and partition merges.

Crash failures are detected by the stack through either a timeout mechanism (which causes a `SUSPECT` notification) or a message from another node (which causes a `SLANDER` notification). In either case, the state machine eventually transitions to the `I_Suspect` state either directly or through the `Slander_Suspect` state. While the `I_Suspect` state may be reached on several nodes of the system, only the state machine at the node that determines itself to be the leader of the group proceeds to the `Leader` state. In the case of partition merges, the state machine at the merging leader transitions to the `Want_to_Merge` state when it receives a notification that the leader wants to merge with another partition. Later in the protocol, the merging leader requests a merge from the primary leader. The receipt of this request by the primary leader causes its state machine to transition to the `Got_Merge_Req` state.

Irrespective of the nature of the event that triggered the membership change, the leader now begins blocking the group by broadcasting a *Block* message to the followers. On receipt of this message, the followers' state machines transition to the `Got_Block` state. At that point, the state machines on both the leader and follower wait for the stack to notify the application, via a *Block* callback, that the group is about to block. When this callback is called, the state machines on the leaders and followers transition to the `L_App_Block` and `F_App_Block` states, respectively, and wait there for the callback to return. Henceforth, the application is not allowed to send any new messages until the group is unblocked. It should be clear that the time spent in application callbacks depends on the application. The workload generator does not do any work in the callbacks and hence represents the best-case scenario in terms of blocking time.

The Prepare Phase

The *Prepare Phase* of the protocol begins when the application returns from the *Block* callback. This phase is an agreement protocol in which the nodes in the group agree upon a *consistent* set of messages that must be delivered in the current view so that virtual synchrony is preserved (any message transmitted in the current view must be delivered in the same view). The `L_Wait_SyncInfo` state on the leader represents the time spent by the leader waiting for each follower's version of the virtual synchrony information. This

information is sent to the leader by each follower after it returns from the *Block* callback. The `F_Wait_SyncInfo` state represents the time each follower then waits for the leader to send out a consistent version of the virtual synchrony information in a *SyncInfo* message. The leader computes this consistent version based on the information received from all of the followers. The dissemination of consistent virtual synchrony information to all the nodes in the group marks the end of the prepare phase.

The Delivery Phase

The next phase of the protocol is the *Delivery Phase*, in which the nodes in the system deliver all undelivered messages from the set of messages agreed upon in the Prepare Phase. The `L_Wait_Delivery` and `F_Wait_Delivery` states represent the time spent by the leader and followers, respectively, in delivering messages. Once a follower finishes its message delivery, it sends a *BlockOK* message to the leader. The `L_Wait_Block` state of the state machine represents the time the leader spends waiting to receive *BlockOK* messages from all of the followers after it has finished its own message delivery. At the end of this phase, no messages in transit exist anywhere in the system, and the group is said to be *blocked*.

The Merge Sub-phase

The *Merge Sub-phase* of the membership protocol is executed only on a merging leader and is entered after the `L_Wait_Block` state. During the Merge Sub-phase, the merging group is blocked and the merging leader sends a *MergeRequest* message to the primary leader, requesting a merge. The `Wait_Merge_Resp` state thus represents the time spent by the merging leader waiting for a response. The primary leader may either deny the request, in which case it informs the merging leader immediately, or accept it, in which case it blocks the primary group. After the primary group is blocked, the merging leader is informed via a *MergeGranted* message, which also contains the new view. Irrespective of the primary leader's response, the merging group undergoes a view change. If the merge is granted, the primary leader becomes the leader of the combined group. Since this phase occurs only in a merging leader, its time does not affect the blocking time for the primary group.

The Create View Phase

This phase begins when the leader enters the `L_Create_View` state of the state machine. This state represents the time spent by the leader in creating a new view, which it broadcasts to the group. The time in the `F_Wait_View` state represents the time spent by each follower waiting for the new view. In the case of partition merges, the primary leader also sends the

view to the merging leader which forwards the view to the merging group. When a node has a copy of the new view, it creates a new stack. The `L_Init_Stack` and `F_Init_Stack` states of the state machine represent the time spent in this operation on the leader and followers respectively. The `FOLLOWER` transition from `L_Init_Stack` to `F_Init_Stack` represents a merging leader's change of status from a leader in the old view to a follower in the new view.

Application State Transfer

After the new stack is created, the group membership protocol is complete and the group can be unblocked. However, in some configurations, Ensemble also provides a facility that enables applications to agree on a common state for the new view. The application is prevented from sending messages during the state transfer, and hence this phase is treated as part of the membership protocol when it is present in the stack (as far as blocking behavior is concerned). Therefore, it is also included in the state machine.

State transfer begins when the stack at each node requests some application state via the *BlockView* application callback, and the followers send their versions to the leader. The state machine represents the time spent in the callback on the leader and followers by the `L_App_BlockView` and `F_App_BlockView` states respectively. The `L_Wait_TMerge` state represents the time the leader waits to receive each follower's version of the state. The leader asks the application to compute a single state from the received information via the *InstallView* callback, and broadcasts the returned state to the followers in the form of a *TView* message. The `L_App_InstallView` state represents the time spent in the callback on the leader, and the `F_Wait_TView` state represents the time spent by each follower waiting for the state returned by the leader. Once a node has a copy of the application state, it informs the application, via the *Unblock* callback, that the view is unblocked, and allows the application to transmit messages from this point onwards.

3.5.2 Multiple-Fault State Machine

Figure 3.4 shows the state machine that models the behavior of the membership protocol (on a single node) in response to correlated crash failures. This state machine contains more edges than the single-fault state machine developed in Section 3.5.1. However, no new states are added, because the membership protocol itself does not change, but only undergoes additional transitions when multiple failures occur. The additional edges were added using the technique described in Section 3.4. The rest of this section describes the additional edges added to each phase of the protocol.

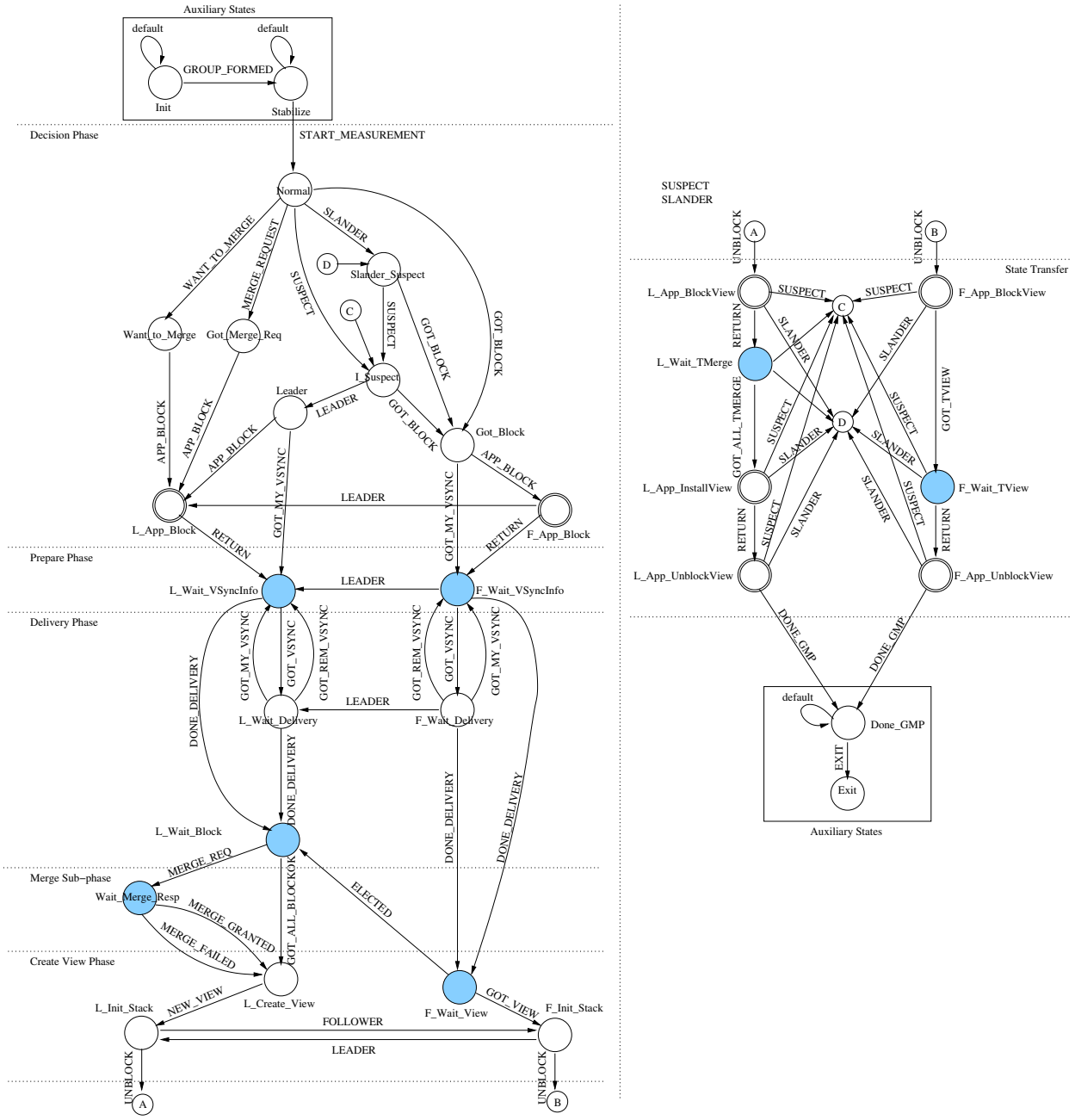


Figure 3.4: State Machine for the Correlated Fault Injections

Auxiliary and Decision Phases

Since the things that need to be done to set up the environment for measurement remain unchanged from the single-fault case, no new incoming or outgoing transitions are added to the auxiliary states. The decision phase remains largely unchanged, with the only changes happening in the manner in which this phase finishes. This is because an additional failure occurring towards the end of the group membership protocol (in the state transfer phase) essentially causes the membership protocol to restart. However, because the application is still blocked in the state transfer phase, it need not be blocked again. Therefore, the *Block* callback can be completely skipped. This is represented by the `GOT_MY_VSYNC` transitions. Also, if a failure occurs while a node is executing the *Block* callback, an election may take place, causing a follower to become a leader. This is represented by the `LEADER` transition from the `F_App_Block` to the `L_App_Block` states.

Prepare and Delivery Phases

If a failure occurs when a node is in the `F(L)_Wait_SyncInfo` or the `F(L)_Wait_Delivery` state, then the node discards the old virtual synchrony information that was agreed upon, and all the nodes re-execute the agreement protocol. This happens when a node receives a new copy if its own or some remote virtual synchrony information. That receipt is represented by the transitions `GOT_MY_VSYNC` and `GOT_REM_VSYNC`, which cause a change of state from the `F(L)_Wait_Delivery` states to the `F(L)_Wait_VSyncInfo` states. If the leader crashes, then one of the followers is elected to be the new leader. The election is represented by the `LEADER` transition from the `F_Wait_Delivery` and `F_Wait_SyncInfo` states to the `L_Wait_Delivery` and `L_Wait_SyncInfo` states, respectively. Finally, even though the nodes re-execute the agreement protocol, the new agreement can only result in a reduction of the messages that need to be delivered (since no new messages can be introduced into the group after it blocks). Hence, if a node finishes delivering the original set of messages (agreed upon in the first execution of the agreement protocol), then the node may get a `DONE_DELIVERY` message even while the second agreement protocol is in progress. That receipt is represented by the `DONE_DELIVERY` transitions in the `F(L)_Wait_SyncInfo` states. Once the leader finishes delivery, it waits for *BlockOK* messages from the followers, as before.

Create View Phase

If a follower failure is detected in the Create View phase, then all the leader does is remove the failed member from the new view before transmitting it (since there are no outstanding messages in the group). Hence, there are no new transitions needed. If the leader fails,

then one of the followers is elected the new leader. However, the new leader has no way of knowing that the old leader had already received *BlockOK* messages from all the followers. Hence, it must solicit them from the followers again. This is represented by the `ELECTED` transition from the `F_Wait_View` state to the `L_Wait_Block` state. Finally, if the failure is detected in the `F(L)_Init_Stack` states, then the stack does nothing, but defers the failure processing until after the stack has been initialized (in the new view). The only thing that happens is that a follower gets elected as the new leader if the old leader has crashed. This is represented by the `LEADER` transition from the `F_Init_Stack` to the `L_Init_Stack`.

State Transfer Phase

Technically speaking, the state transfer phase is not really a part of the core group membership protocol. Hence, if a crash is detected during this phase, the membership protocol must start all over again. However, the only thing that differentiates a crash in this phase from a crash during normal operation is that the application is still blocked in this phase. Hence, the *Block* callback is not executed for crashes detected during this phase. The `SUSPECT` and `SLANDER` transitions originating from all the states in this phase model this fact.

3.6 Fault Triggers

This section describes the fault triggers used to inject the single and correlated faults into the system. These triggers are Boolean-valued functions whose domain is the global state of the system as defined by the node state machines.

3.6.1 Single-Fault Triggers

For the single-fault experiments, a single fault is injected into the system when a group of the proper size has formed and each state machine has transitioned to the `Normal` state of the state machine. For crash failure injections, the node NodeFail is injected. The fault method for this fault simulates a crash by generating a segmentation failure exception. For node join injections, the node Node_1 is injected. The fault method for the node join starts up a new node NodeMerge on an unused host via the `ssh` command and using the dynamic node invocation facilities of Loki [CCLS00]. The newly created node automatically finds the main group and attempts to merge with it, effectively injecting a group merge into the system.

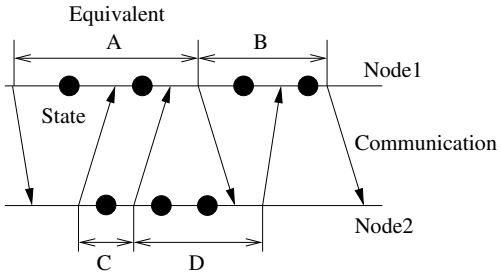


Figure 3.5: Computation of Equivalent State for Failure Injection

The triggers for single crash failure injections and single node joins are, respectively:

Crash Failure Injections
At NodeFail :
$(\text{Node}_1 : \text{Normal}) \wedge \ldots \wedge (\text{Node}_n : \text{Normal}) \wedge (\text{NodeFail} : \text{Normal}) \mapsto \text{CrashFault}$
Node Join Injections
At Node_1 :
$(\text{Node1} : \text{Normal}) \wedge \ldots (\text{Node}_n : \text{Normal}) \mapsto \text{InjectNodeJoin}$

3.6.2 Correlated Fault Triggers

For the correlated fault experiments, exactly two faults are injected in every experiment. The first fault is injected when the group is formed and all nodes are in their `Normal` state. Hence, the trigger for this fault is the same as for the single-fault experiments. The second fault is then injected when the system is in certain specific states during the execution of the group membership protocol. The smallest set of local states that need to be targeted by the second crash failure injection for completeness can be computed as follows. A node crash impedes the progress of the protocol whenever the rest of the group waits for a protocol message from the crashed node. If the execution path of a node is split into intervals delimited by the times at which it transmits protocol messages to the rest of the group, then crashes in each distinct interval will prevent distinct protocol messages from being sent out; thus, each crash can potentially have a different impact on the other nodes in the system. Therefore, at least one state from each interval must be targeted. This is demonstrated in Figure 3.5, in which the intervals A and B are formed on Node1 and the intervals C and D on Node2 due to their pattern of communication.

A crash anytime before the end of an interval prevents propagation of any of the node's actions in that interval to the rest of the group. Hence, it is assumed that the response of the group to a fault injection in a state is representative of the response to injections in other

states in the same interval. Thus, all the states in an interval form an equivalence class, and a single injection per interval is sufficient. However, this argument does not mean that the response to a crash at any instant could be exactly computed based on the response to a crash in the interval containing that instant, since other factors in the protocol (such as expiration of timeouts) are sensitive to the exact time at which a crash took place.

The dotted edges in the node state machine in Figure 3.3 represent protocol messages. The states between successive outgoing protocol messages represent the intervals. For example, the dotted edges representing the *Send(BlockOk)* and *Send(TMerge)* protocol messages in the state machine define an interval that contains the `F_Wait_View`, `F_Init_Stack`, and `F_App_BlockView` states.

3.6.3 Correlated Fault Trigger Implementation

As seen in the previous section, most correlated crash failure injections involve injecting the second fault into a node when it is in a particular local state (or in one of a set of local states) and no other injections have been done. Let S be this set of local states in which the fault is to be injected. If the injection is to be done in a leader, S will contain only leader states ($\mathcal{L}$) whose names have the prefix `L_`. Since any node in a system can be a leader, fault triggers must be installed on each node in the system. These triggers must also be temporally synchronized; i.e., after a leader has crashed and the group has elected a new one, the same trigger should not be invoked a second time on the new leader. We ensure that it is not by checking that when the fault is injected, no nodes except NodeFail (which is always the target for the first crash failure injection) have crashed.

If the injection is to be done in a follower, S must contain only follower states ($\mathcal{F}$) that have the prefix `F_`. In that situation, fault triggers need to be installed only on two nodes since at least one of them must be a follower at any time. The triggers must be spatially synchronized (i.e., only one of the triggers injects the fault in a given experiment, even if both nodes are followers). To achieve this spatial synchronization, a set function $L(S) : 2^{\mathcal{F}} \mapsto 2^{\mathcal{L}}$ is defined. This function, when given a set S of follower states that are to be targeted, returns the subset of leader states ($\mathcal{L}$) that the leader of a group *could* be in when a follower is in one of the S states. Then, one of the two nodes on which the fault triggers are specified is designated a primary node, and the other one a secondary node. The trigger on the primary node is made to fire when the primary is in one of the states in S. The trigger on the secondary node is made to fire only when the primary node is in one of the states in $L(S)$ (which means that it is a leader) and the secondary node is in one of the states in S.

Table 3.1: Local State Selection for Correlated Injections

S	**L(S)**	**Protocol message blocked**
{Leader}	n/a	Block Message
{Got_Block}	{Leader,L_App_Block, L_Wait_SyncInfo}	Local Sync Info
{L_Wait_SyncInfo}	n/a	Global Sync Info
{F_Wait_Delivery}	{L_Wait_Delivery,L_Wait_Block}	Completion of Msg Delivery
{L_Create_View}	n/a	New View
{L_Wait_TMerge, L_App_InstallView}	n/a	Agreed Appl. State (TView)
{F_Init_Stack, F_App_Block_View}	{L_Init_Stack,L_App_BlockView, L_Wait_TMerge}	Local Appl. State (TMerge)

The resulting generic fault triggers are as follows:

Leader Crash Injections
$\forall i = \{1, \ldots, n\}$, At Node$_i$:
(Node$_i$: state $\in S$) $\wedge \neg$(Node$_1$: CRASH) $\wedge \ldots \wedge \neg$(Node$_n$: CRASH) $\mapsto$ CrashFault
Follower Crash Injections
At Node$_{primary}$: (Node$_{primary}$: state $\in S$) $\mapsto$ CrashFault
At Node$_{backup}$: (Node$_{primary}$: state $\in L(S)$) $\wedge$ (Node$_{backup}$: state $\in S$) $\mapsto$ CrashFault

Note that both temporal and spatial synchronization of crash failures require knowledge of global state. Hence, the fault triggers serve as a good example of the use of global state information in fault injection. The sets of states S in which fault injections are to be done (according to the state selection algorithm presented in Section 3.6.2) are shown in Table 3.1. The table also shows the corresponding leader sets $L(S)$.

3.7 Measures

This section describes the measures that were used to quantify the effects of the group membership protocol on the availability of the group. One possible approach would have been to measure the availability of the system directly. However, since availability is strongly correlated with the workload, the workload would have to be representative for availability measurements to have any relevance. It is impossible to argue for a single "representative" workload for all the possible uses of a group communication system. Moreover, availability by itself does not provide insight into the causes of downtime, and is of limited use from the

point of view of improving a protocol. Hence, the measure that was used was the *blocking time*, which is a conditional measure that is conditioned on the workload and other system parameters (such as group size).

3.7.1 Single Node Blocking Time

Blocking time was defined in Section 3.1 in terms of a state machine model of the role of a group membership protocol. Briefly, it is the amount of time which the group blocks message transmission. Blocking time one of the most important components of unavailability in a group communication system. For this study, the blocking time due to a group membership change was the primary measure of interest. It is relatively straightforward to define blocking time for a single node. This can be done by examining the state machine for the node, identifying the states during which message transmission/receipt is blocked, and adding together the time spent in all such states.

Since an Ensemble application may or may not use the state transfer protocol provided, the blocking times for two versions of the membership protocol are defined. The version that includes the application state transfer function (Section 3.5.1) is called the *Full Protocol*, and the version without it is called the *Core Protocol*. The blocking times for the Core Protocol can be obtained by treating the state transfer phase as non-blocking and as a part of the application. The last states in which applications are allowed to transmit are the `L(F)_App_Block` states of the Decision Phase. The group unblocks in the `L(F)_App_BlockView` states for the core protocol and in the `L(F)_App_UnblockView` states for the full protocol. The sets B_{full} and B_{core} of blocking states in the full protocol and the core protocol, respectively, are defined as follows:

B_{core}	=	{L_Wait_SyncInfo, L_Wait_Delivery, L_Wait_Block, Wait_Merge_Resp, L_Create_View, L_Init_Stack, F_Wait_SyncInfo, F_Wait_Delivery, F_Wait_View, F_Init_Stack}
B_{full}	=	$B_{core} \cup$ {L_App_BlockView, L_Wait_TMerge, L_App_InstallView, F_App_BlockView, F_Wait_TView}

3.7.2 Group Blocking Time

To get an estimate of the group blocking time, it is not sufficient to merely average the individual blocking times on each node. This is because such averaging results in a loss of information about the time of blocking on a node relative to the other nodes. For instance, the situation where two nodes block from time t to time $t + T$ would be clearly better than the situation where one node blocks from time t to $t + T$, and the second one blocks from

time $t+T$ to $t+2T$ even though the average blocking time is the same in both instants. However, defining a common measure of blocking that is useful for all applications is difficult, because different applications have a different notion of availability. For example, an online messaging system may be considered to be available only when all its nodes are available, whereas a replication scheme may be available if at least one node is available.

To allow quantification of blocking time in face of such diverse needs, a function "Block-Ratio" is defined on the group. If BlockRatio returns p at time t, then $(100 \times p)\%$ of the nodes in the group are blocked at that time. The predicate *p-Blocked* is True when BlockRatio=p. Hence, if all the nodes in the group are blocked at some time, then the predicate 1-Blocked is True. If no nodes in the group are blocked, then 0-Blocked is True. If some node in the group is blocked (but it is not known how many), then 0-Blocked is False. The two situations of either all nodes being being blocked, or at least some nodes being blocked are expected to be very common. Hence, a group is said to be *Totally Blocked* when 1-Blocked is True (no messages can originate anywhere in the system), and *Partially Blocked* when 0-Blocked is False (not all nodes may generate messages).

For a given set of blocking local states B, the function BlockRatio can be defined on the global state of the system using the Loki measures language as follows:

$$
\begin{aligned}
\text{NumBlocked}(B,t) &= \text{how\_many}(SMLIST, [x, \neg\text{chk\_state}(x, \text{CRASH})], \\
&\qquad [x, \text{if\_any}(B, [y,1], [y, \text{chk\_state}(x,y)])]) \\
\text{NumNodes}(t) &= \text{how\_many}(SMLIST, [x, \neg\text{chk\_state}(x, \text{CRASH})], [x,1]) \\
\textbf{BlockRatio}(B,t) &= \frac{\text{NumBlocked}(B,t)}{\text{NumNodes}(t)}
\end{aligned}
$$

Using the function BlockRatio, the predicates for p-Blocked, Totally-Blocked, and Partially-blocked can be defined using the measures languages as follows:

$$
\begin{aligned}
\text{p-Block}(B,t) &= \text{BlockRatio}(B,t) == p \\
\text{Partially-Blocked}(B,t) &= \neg(\text{p-Block}(B,t) == 0) \\
\text{Totally-Blocked}(B,t) &= \text{p-Block}(B,t) == 1
\end{aligned}
$$

The lengths of time for which Partially-Blocked and Totally-Blocked are True for some $B \in \{B_{core}, B_{full}\}$ are called $t_{partial}(B)$ and $t_{total}(B)$ respectively. The results presented in Chapter 4 compute these two quantities for both the full protocol (B_{full}) and the core protocol (B_{core}). Application of the Partially-Blocked and Totally-Blocked predicates to the global timeline for an experiment yields predicate timelines that are Boolean-valued

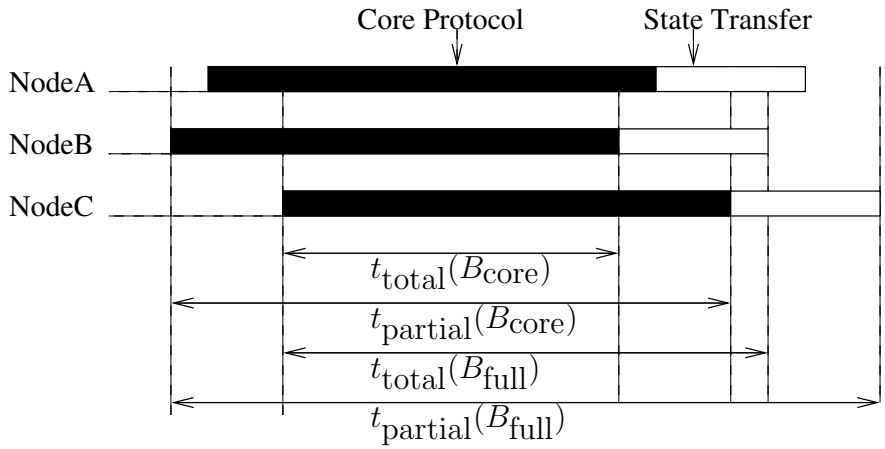


Figure 3.6: Relationship Between Various Block Predicates

functions of time. The intervals for which these predicate timelines are True (as computed by the observation function total_duration(True, Exp_Begin_Time, Exp_End_Time)) give the times t_{total} and $t_{partial}$, respectively.

The definitions of the Partial-Blocked and Totally-Blocked predicates and the full and core protocols impose several constraints on the intervals. Figure 3.6 shows the relations between the intervals. It shows the execution of the core and full protocols on three nodes, and the definitions of the block intervals over the nodes. First, any total interval is properly contained within its corresponding partial interval ($t_{total}(B) < t_{partial}(B)$). This difference is due to the fact that nodes enter and leave the protocol at slightly different times because of differences in message delay, speed, load, and protocol characteristics. The difference between the start times of a partial and a total interval is called the *preblock stagger* and represents the maximum blocking time spent by a group member waiting for other nodes to start the protocol. Similarly, the difference between the end times of a partial and total interval is called the *postblock stagger*. The postblock stagger for the core protocol represents the maximum time spent by a node waiting for another node to finish the core protocol, and begin state transfer. Second, the total and partial blocking intervals for the core protocol start at the same time as the corresponding intervals for the full protocol, but are properly contained within them. Therefore $t_{total}(B_{core}) < t_{total}(B_{full})$ and $t_{partial}(B_{core}) < t_{partial}(B_{full})$. Finally, the preblock stagger is the same for both the core and full protocols.

3.7.3 Individual State Holding Time Ratios

To determine the parts of the protocol that most strongly affect the blocking time of a group, measures that calculate the average contribution of each individual state (from the set of

Table 3.2: Computing the Contributions of Individual States to Blocking Time

Ratio of Individual State Holding Time at Node$_i$ to p-Block(B) Interval	
Tuple	PBLOCKTIME
Predicate Fn	p-Block(B, t)
Observation Fn	total_duration(TRUE, Exp_Begin_Time, Exp_End_Time)
Filter Fn	PBLOCKTIME$\neq$0
Tuple	RATIO
Predicate Fn	p-Block$(B, t) \wedge$ chk_state(Node$_i$, state)
Observation Fn	total_duration(TRUE, Exp_Begin_Time, Exp_End_Time)/PBLOCKTIME
Filter Fn	True

blocking states) to the group blocking time were also computed. The average contribution of a state to the blocking time is defined as the ratio, of the blocking time spent in that state (averaged over all nodes), to the total blocking time. The Loki measure that computes that ratio for a single node (Node$_i$) is as shown in Table 3.2. Computing the average ratio for a state requires multiple study-level observations, one for each node in the system, which are then averaged using a campaign-level measure. The particular blocking time used (partial or complete, and over the full protocol or the core protocol) depends on the state whose contribution is being computed. If the state is part of the state transfer protocol, then the full protocol blocking times are used. Otherwise, the core protocol blocking times are used. Hence, any excessively long state transfer times do not overshadow the contributions of the other parts of the protocol. Although either Partially-Blocked or the Totally-Blocked predicates could be used as the particular p-Block predicate, the results presented in Chapter 4 used Partially-Blocked. Thus, the pre-block and post-block stagger is also included in the denominator, and hence the the individual state contributions may not sum to 1 (the difference is the contribution due to synchronization delays).

3.7.4 Removal of Effect of Timeout

During the correlated fault experiments (Section 4.2), it was observed that the timeout interval used to detect crashes dominated the blocking times. To gain better insight into the protocol, it was necessary to remove the dominant effect of the timeout. Reducing the timeout interval to the point where it did not dominate the results, (i.e., reducing it to the order of a few milliseconds) was not feasible. That was because the reduction would have led to false alarms (spurious crash detection), and caused the membership protocol to be initiated even when there was no crash. Hence, timeout interval was factored out of the blocking time analytically through the Loki measures language. We did so by examining all

Table 3.3: Removal of the Effect of a Timeout Interval

Measure to Remove the Effect of a Timeout from a p-Block Interval	
Tuple	**SECOND_CRASH_TIME**
Predicate Fn	if_any($SMLIST$, [$x, x \neq$ NodeFail], [x, chk_event(x, CRASH)])
Observation Fn	instant(UP, IMPULSE, 1, Exp_Begin_Time, Exp_End_Time)
Filter Fn	chk_fault(NodeFail, CRASH, $CORRECT$)∧ chk_fault({$\text{Node}_1, \text{Node}_2, \ldots \text{Node}_n$}, CRASH, $INJECTION$)]) == 1∧ chk_fault({$\text{Node}_1, \text{Node}_2, \ldots \text{Node}_n$}, CRASH, $CORRECT$)]) == 1∧
Tuple	**FIRST_SUSPECT_TIME**
Predicate Fn	if_any($SMLIST$, [$x, x \neq$ NodeFail], [x, chk_state(x, CRASH)])∧ if_any($SMLIST$, [x, 1], [x, chk_event(x, SUSPECT)])
Observation Fn	instant(UP, IMPULSE, 1, Exp_Begin_Time, Exp_End_Time)
Filter Fn	True
Tuple	**LAST_PROGRESS_TIME**
Predicate Fn	if_any($SMLIST$, [$x, x \neq$ NodeFail], [x, chk_state(x, CRASH)])∧ if_any($SMLIST$, [x, 1], [x, if_any($EVENTLIST$, [y, 1], [y, chk_event(x, y)])
Observation Fn	instant(DOWN, ALL, -2, Exp_Begin_Time, FIRST_SUSPECT_TIME)
Filter Fn	True
Tuple	**BLOCK_TIME**
Predicate Fn	p-Block(B, t)
Observation Fn	Timeout = total_duration(TRUE, **max{**LAST_PROGRESS_TIME, SECOND_CRASH_TIME**}**, FIRST_SUSPECT_TIME) **return** total_duration(TRUE,Exp_Begin_Time,Exp_End_Time)-Timeout
Filter Fn	True

the events in the system between the second crash failure injection and the first `SUSPECT` event subsequent to it. The time between the `SUSPECT` event and the event just prior to it (which was the last event due to which the protocol made progress before waiting for the crashed node to perform some action) is the time the protocol spends waiting for a timeout. The overlap of this interval with the group blocking time predicate gives the contribution of the timeout to the group blocking time. Subtracting the duration of the overlap from the group blocking time removes the effects of the timeout. The Loki measure definition which performs the removal and computes the p-Block interval with the timeout factored out is as shown in Table 3.3.

The measure is composed of 4 tuples. The first tuple, named SECOND_CRASH_TIME, performs two functions. First, it retains only those experiments in which the first crash failure was injected correctly in node NodeFail, the second crash failure was injected exactly once into the system, and that injection was correct. The tuple filters out all other experiments. The second function performed by the SECOND_CRASH_TIME tuple is to compute

the time at which the second crash occurred. It does so via the predicate and observation functions. The "instant" function used in the observation function computes the time of the first impulse on the predicate timeline. The FIRST_SUSPECT_TIME tuple compute the time at which the first failure suspicion was generated after the second crash injection. The LAST_PROGRESS_TIME tuple computes the time of the last event after the second crash failure, and before the occurrence of the first suspicion event. The -2 argument used in the instant function indicates that the time of occurrence of the second last impulse (the last impulse is the suspect event itself) in the interval [Exp_Begin_Time, FIRST_SUSPECT_TIME] is to be computed. Finally, the BLOCK_TIME tuple computes the difference between the time the p-Block predicate is True and the contribution of the timeout. The contribution of the timeout is computed as the duration of time the p-Block predicate is True starting from maximum of the time of the last event occurrence and the time of the second crash, and ending at the time of the first suspicion event.

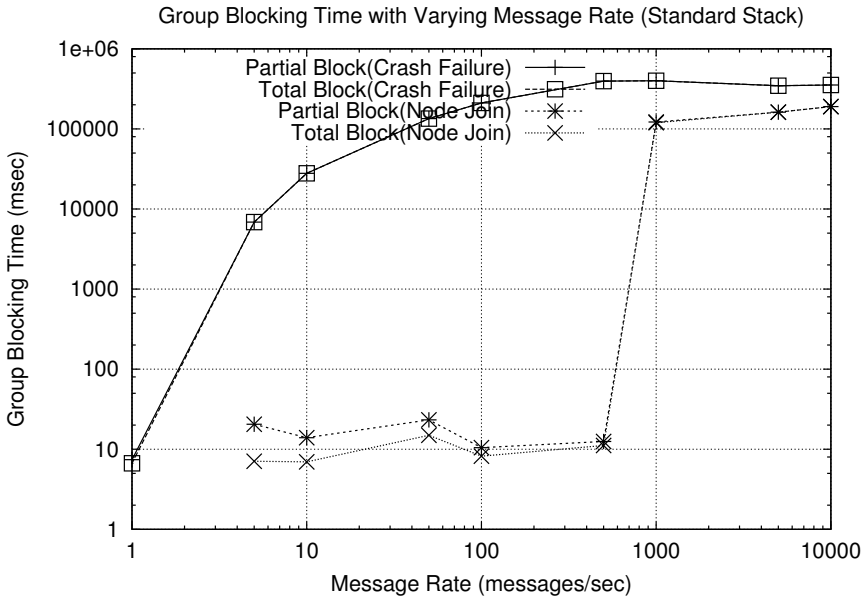
Chapter 4

Results

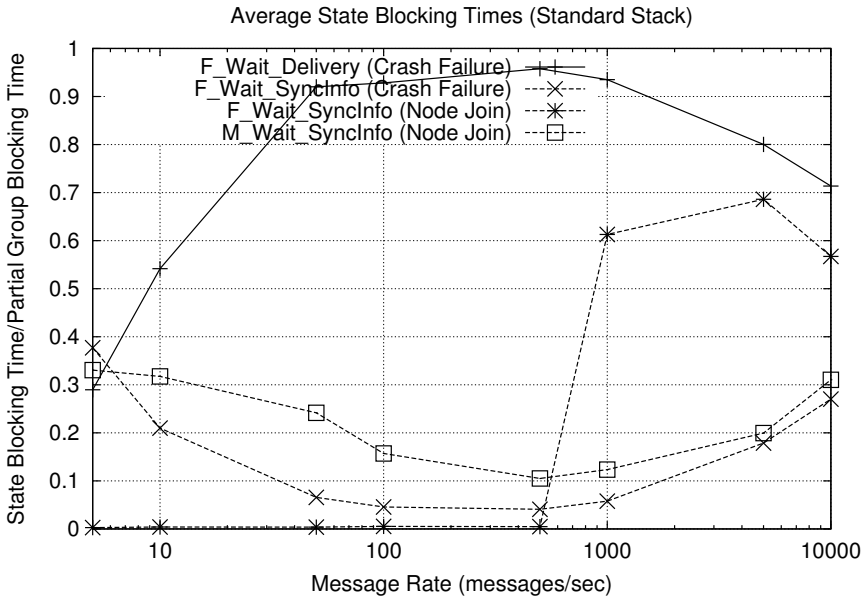
This chapter presents the measures extracted from the fault injection experiments specified in Chapter 3. The experiments are divided into two categories. The first set of experiments is for the single-fault injections with varying workloads and group sizes. The second set of experiments is for the correlated-fault injections with a fixed workload and group size. Finally, this chapter concludes with a discussion on the lessons learnt from the study. The experimental setup used for the study consisted of a testbed with 12 identical hosts running Redhat Linux 6.2. Each machine had a Pentium III 1GHz processor and 256MB RAM. The bytecode version of the Ensemble 1.20 distribution compiled with the OCaml 3.02 compiler was used for the study. Each application node was started on a separate host. A separate control host was used to run the Loki manager and analyze the results.

4.1 Single-Failure Experiments

In the single-failure experiments, a single fault was injected into the system when all the nodes were in the `Normal` state. One of the environmental parameters, i.e., the workload or the group size, was varied in each set of experiments while the other was kept constant. After the experiments were conducted, we observed that message blocking depended on whether the first crash was a leader or not. For leader crashes, the group cannot deliver messages for the entire duration of the crash detection timeout period. For follower crashes, the issue is more subtle. Hence, we have filtered out experiments in which the first crash was a leader, and present measures for follower crashes only.



(a) Complete Group Blocking Time



(b) Individual State Holding Times

Figure 4.1: Blocking Times for a Standard Stack with Varying Workload

4.1.1 Effect of Workload Variation on an Unmodified Stack

During these experiments, a follower crash failure and a node join were injected in a group of fixed size and the workload rate parameter λ was varied from 1 message/sec to 10,000 messages/sec. The initial group size was five nodes before the crash for crash failures, and four nodes before the join for node joins. Hence, in both cases, four nodes participated in the core membership protocol. Figure 4.1(a) shows the resulting total and partial blocking times for the full protocol. Even though the protocols for crash failures and node joins are not very different, the blocking times for a crash failure are five orders of magnitude higher than those for node joins for low message rates. In addition, as the message rate increases, the blocking time for node joins increases suddenly and becomes comparable to the blocking time for crash failures.

The phenomenon of the sudden increase in blocking time for node joins (at high message rates) was investigated by examining the global timelines. They revealed that the following sequence of events was occurring at high message rates. The joining node would block its group, and make a request to the leader of the primary group to perform the merge. However, due to the high message volume in the primary group, by the time the primary leader finished blocking the primary group and sent a merge accepted message to the joining node, the joining node had already timed out waiting for the merge. As a result, the joining node discarded the *MergeGranted* message from the primary leader when it received one (thinking that the message was a spurious message). However, the primary group still continued with the merge process, assuming that the joining node was now part of the primary group. Hence, the protocol blocked in the state transfer phase while waiting for a response from this node, and finally timed out, leading to the removal of the "faulty" join node from the group. Hence, the behavior was similar to that of a crash failure. This situation could have been avoided if the joining node had responded to the primary leader's *MergeGranted* message to indicate that it had aborted the merge. This suggests that a negative acknowledgment protocol for group membership messages would be a useful enhancement to the group membership protocol.

It is interesting to note that the reason that the primary leader took so long to respond to the merge request was because of the large message volume in the primary group coupled with the FIFO event processing in the Ensemble stack. Due to the FIFO property, important group messages must wait alongside with normal messages for processing at each layer. This can cause problems such as the one described above. Using priority queues as the event processing model for the stack (and assigning higher priorities to important protocol messages) would have also solved the above problem. Hence, a case can be made against

using a strict FIFO stack based model for group communication applications with high expected workload coupled with high availability requirements.

To analyze the phenomenon of the extraordinarily large blocking times for crash failures, the ratios of the time a node spent in each blocking state to the partial group blocking time were plotted. Figure 4.1(b) shows these ratios for the `F_Wait_Delivery` and `F_Wait_SyncInfo` states for both crash failures and node joins. It can be seen that for crash failures, the `F_Wait_Delivery` state forms a significant fraction of the time spent blocking. However, for node joins with message rates up to 500 messages/sec, the `F_Wait_Delivery` state accounts for a very small fraction of the blocking time. This reflects a problem with message delivery in the presence of crash failures. Addition of additional message transmit and receive events to the state machine and an examination of the resulting global timelines revealed that messages were getting blocked and buffered for most of the `Normal` state. The blocking turned out to be because of interactions of the flow control layer with the failure detection mechanism.

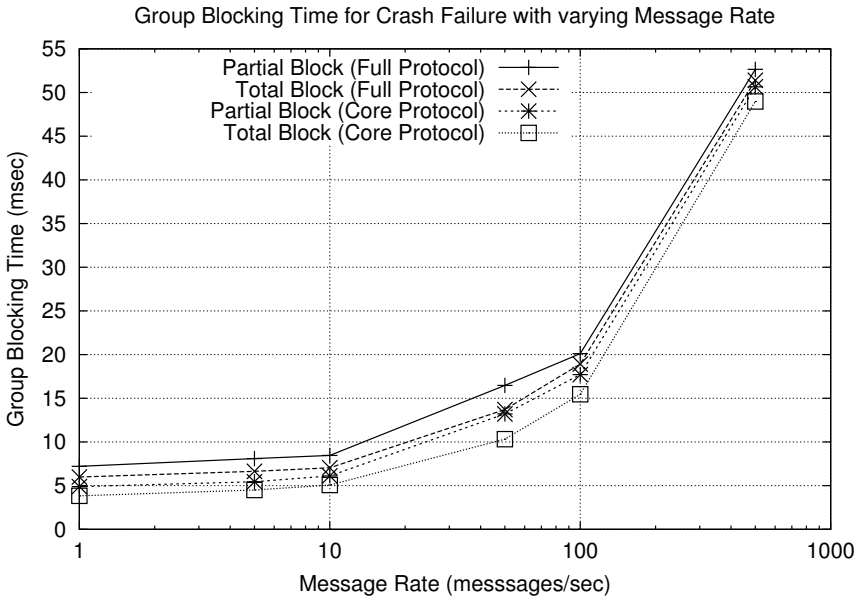
The Ensemble flow control layer *MFlow* for broadcast traffic employs a credit-based scheme. The credit based scheme assigns "credits" to each node. When a node sends a broadcast message to the group, the credits available to the node are reduced. When a node receives an acknowledgement that a message it had previously sent was delivered by all the nodes in the group, the credits for that node are increased. When a node runs out of credits, it is not allowed to transmit any new messages. This scheme is designed to bound the number of unacknowledged messages, thus ensuring bounded buffer sizes at each node (unacknowledged messages must be buffered in case they need to be retransmitted). After a node crash, the flow control layers on the surviving nodes quickly run out of credits because of the lack of message acknowledgments from the crashed node. Hence, subsequent transmissions are buffered by the stack until the crash is detected. At high messages rates, the nodes run out of credits very quickly, and are unable to transmit for almost the entire duration of the failure detection timeout period. Once a failure is detected, the messages acknowledged by all the nodes except the failed node can be thrown away, thus restoring the credits at each node. However, before a view change can occur, the large backlog of messages that were generated by the application at each node (and buffered by the stack) must be delivered (to preserve virtual synchrony). It is this delivery that causes the large amounts of time spent in the `Wait_SyncInfo` and `Wait_Delivery` states, leading to the large blocking times.

A logical solution is for the flow control layer to prevent the application from transmitting when it runs out of credits. After some investigation, it was found that Ensemble does provide a new application interface (though not the one used in this study) that does so.

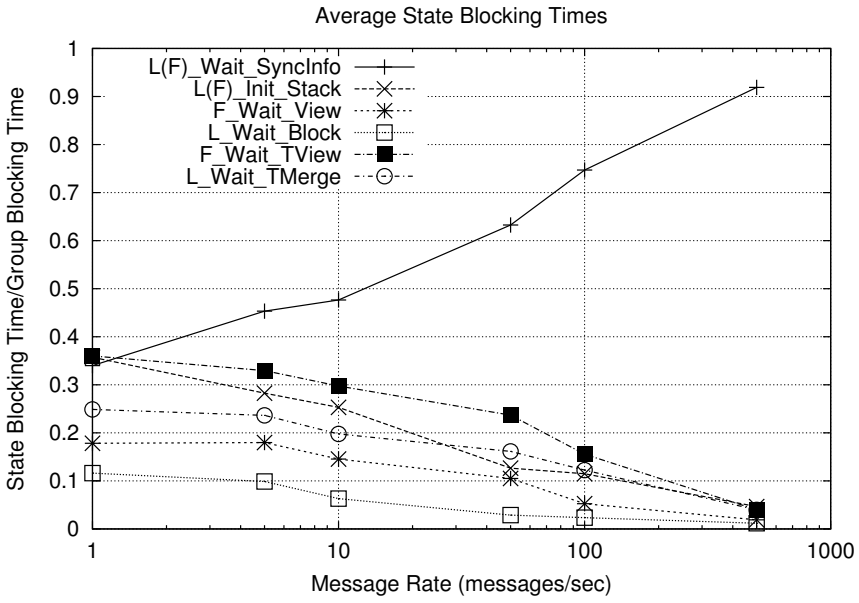
However, this solution is not ideal from an availability standpoint, because it causes the group to be blocked for almost the entire crash detection timeout interval even when all nodes except the crashed node have acknowledged most messages. Alternatively, the credits available to each node can be increased such that there are enough to sustain message delivery at the expected workload throughout the crash detection timeout period. However, doing so couples flow control settings with the timeout interval and expected workload; that is undesirable, since the flow control settings should depend only on network and host capacities. A flow control scheme that takes into account the timeout interval when setting a bound on the maximum number of undelivered messages, but limits the instantaneous rate of traffic according to network and host characteristics, seems a more suitable candidate for high-availability applications. Another alternative is to use an adaptive flow control scheme. In such a scheme, each node would maintain estimates of the sink capacity of the other nodes in the group (in messages/second). The sink capacity would be estimated using the rate at which acknowledgements were received from that node. If the sink capacity of a node changed substantially over a small period of time, e.g., due to a crash, the node would assume that it was a transient error which would be fixed soon (e.g., by the GCS removing the crashed node from the group) and would not reduce the rate of outgoing messages. However, implementing and validating such a scheme would require additional work and is beyond the scope of this thesis. More broadly, investigation of flow control schemes for highly available group communication based applications remains an important area for future work.

4.1.2 Single Crash with Varying Workload

As demonstrated in Section 4.1.1, the flow control layer has a dominating effect on blocking behavior of a standard Ensemble stack using the compatibility application interface. To ensure that the flow control layer did not dominate the behavior of the membership protocol in further experiments, we did our subsequent experiments without a flow control layer in the stack. If message traffic is kept low enough that the flow control layer would not have blocked any messages during failure-free operation, this configuration is equivalent to having a flow control scheme that does not block messages to members during a crash detection timeout period, and just buffers messages unacknowledged by the crashed node. Hence, the workload message rates were restricted to a maximum of 500 messages per second per node. Figure 4.2(a) shows the total group blocking times for the core and full protocols for follower crash failures under a varying workload. The difference between the total blocking times for the full and core protocols is the time taken for application state transfer. It can be seen that the blocking times for crash failures increase linearly with increasing message rate and that



(a) Group Blocking Times



(b) Individual State Holding Times

Figure 4.2: Single Crash-failure Under Varying Workload for a Stack Without Flow Control

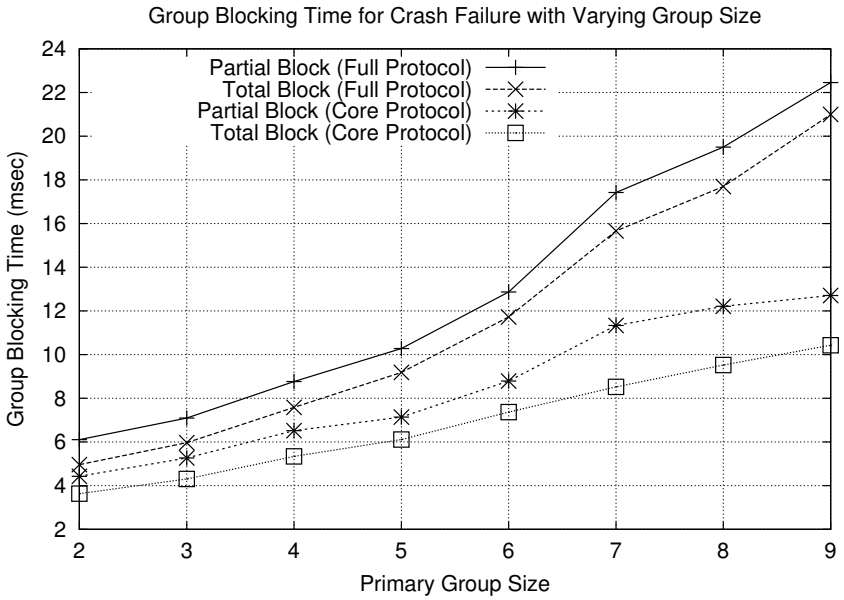
all four curves are fairly close to each other. This indicates that the preblock stagger time ((see Section 3.7.2) is low. This means that not much time is spent by any node waiting on another node to initiate the membership protocol. Figure 4.2(b) shows the contributions of individual states to the overall blocking time for those individual states that have significant contributions. The holding times for the state transfer protocol states (`F_Wait_TView` and `L_Wait_TMerge`) are expressed as fractions of the full protocol partial block time, while the other times are expressed as fractions of the core protocol partial block time. It can be seen that at low rates, a node spends around 35% of its blocking time in the state transfer protocol. The `L(F)_Wait_SyncInfo` states are the largest consumer of the remaining time (the core protocol blocking time). Those states are also primarily responsible for the increase in overall blocking time with increasing workload. At 500 messages/sec, the state transfer protocol consumes only 5% of a node's blocking time, while the `L(F)_Wait_Sync_Info` states consume over 85%.

An investigation of this phenomenon using the global timelines revealed that most of the time spent in the `L(F)_Wait_SyncInfo` states was due to the failure of virtual synchrony information that had been sent by the followers at the beginning of the `F_Wait_SyncInfo` state to reach the leader stack quickly enough. That, in turn, was due to processing going on within one of the layers in the leader's stack; the processing was preventing the single-threaded leader stack from reading the network for new messages. The troublesome processing turned out to be the garbage collection of unacknowledged messages sent to the crashed node. This garbage collection was initiated when the crashed node was declared as Failed at the beginning of the `L_Wait_SyncInfo` state and was a result of allowing message delivery during the crash timeout period by removing the flow control layer. However, note that the penalty of additional garbage collection is much smaller than the penalty of restricting message flow during the timeout interval, as demonstrated in the earlier experiments. This phenomenon suggests the existence of a trade-off. When a crash failure occurs, a flow control layer may either block messages from being transmitted to any member in the group until the failure has been resolved, or allow transmission to live members of the group but face the issue of maintaining and disposing of potentially large unacknowledged message buffers. The former approach guarantees bounded buffers and garbage collection times, but sacrifices availability of the entire group during the crash detection timeout period. Additionally, if the messages that have been blocked need to be buffered by the application anyway, that approach buys nothing. The latter approach increases the availability of the system, but can place on the communication stack the burden of maintaining and disposing of buffers that may be as large as the product of the maximum throughput of the system and the crash detection timeout interval.

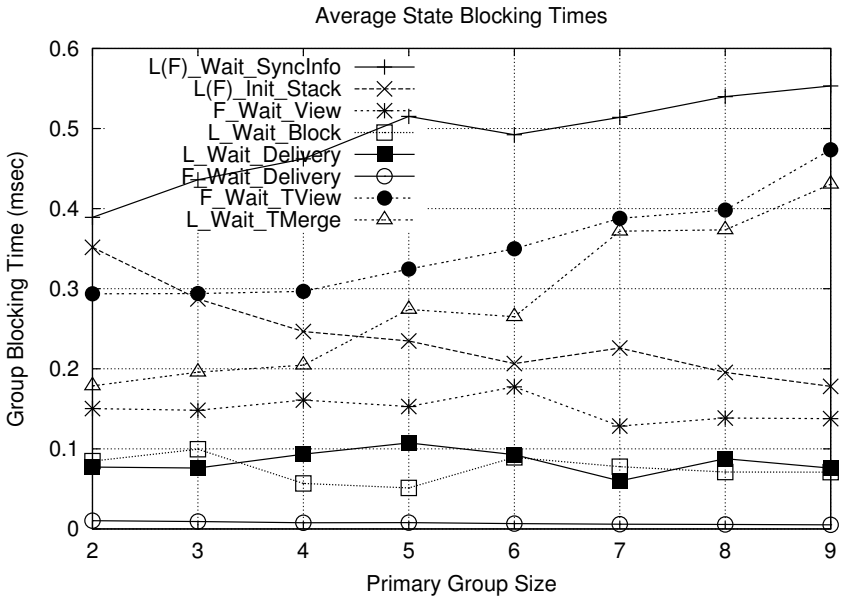
4.1.3 Single Crash with Varying Group Size

During experiments with a single crash-failure injection and varying group size, the message rate was fixed at a rate of 10 messages/sec. Our intention was to keep it low to minimize the garbage collection effects explained earlier, while still ensuring enough load to ensure a non-trivial case. The group size before the crash was varied from 3 nodes to 10 nodes, and a single crash failure was injected. The resulting group blocking times are shown in Figure 4.3(a). It can be seen that all the blocking times increase linearly with group size. The total blocking times for both the full and core protocols are close to their corresponding partial blocking times, indicating that the preblock stagger is low. Hence, nodes are not blocked for too long waiting for other nodes to enter the protocol. Figure 4.3(b) shows the contributions of the individual states to the blocking times. As before, the contributions of the two state transfer protocol states are expressed as fractions of the full protocol partial blocking time, while the contributions of the other states are expressed as fractions of the core protocol blocking time. The increasing ratios for the `L_Wait_TMerge` and `F_Wait_TView` states demonstrate that the state transfer protocol is more sensitive to increasing group size than the core protocol is; this is probably due to the fact that the cost of totally ordered message delivery (required in the state transfer protocol) is higher than the cost of unordered message delivery (which is used in the core protocol). The time the leader spends collecting state information from the followers (`L_Wait_TMerge`) increases more rapidly than the time followers wait to receive a reply from the leader (`F_Wait_TView`). The reason is that an increasing number of nodes implies increased synchronization costs, whereas the time required to compose a global state remains more or less constant.

Of the remaining part of the blocking time, the percentage contribution of the stack initialization states (`L(F)_Init_Stack`) reduces with group size, reflecting a near constant stack initialization time. The percentage contributions of all the other states, except the `L(F)_Wait_SyncInfo` states, remain fairly constant with group size, indicating that they are all equally sensitive to group size. The percentage contributions of `L(F)_Wait_SyncInfo` states, however, show a modest increase with increasing group size. The reason is that as group size increases, the total message traffic in the system increases, because of the constant workload at each node; hence, the increase in the `L(F)_Wait_SyncInfo` state holding time is due to the effects of garbage collection (as described before). It could be argued that this time is an artifact of the flow control scheme (or the lack thereof) and workload, and hence should not be considered part of the blocking time for this set of experiments. However, the alternative would have been to conduct experiments with zero workload, which, in addition to not being very realistic, would also not exercise the virtual synchrony layers at all. If



(a) Group Blocking Times



(b) Individual State Holding Times

Figure 4.3: Single Crash-failure Under Varying Group Size for a Stack Without Flow Control

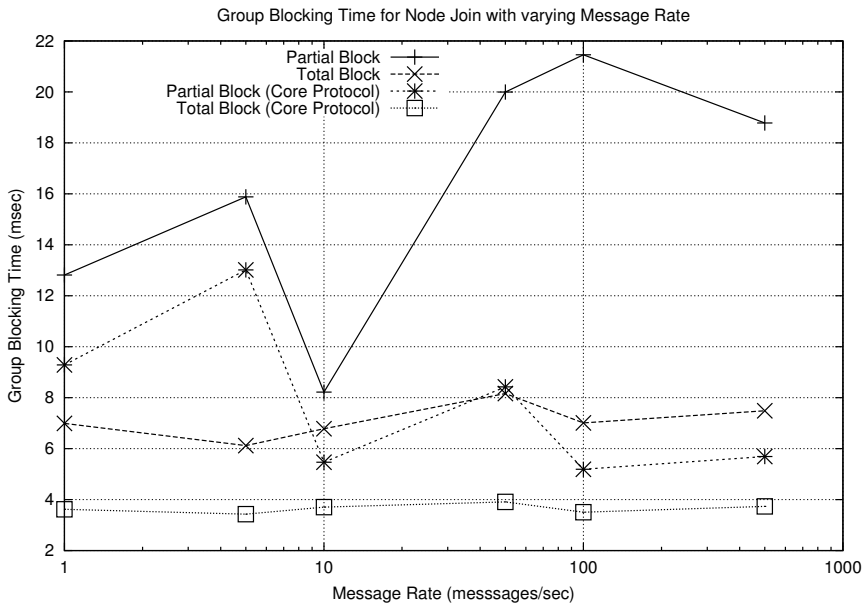


Figure 4.4: Blocking Time for Single Node Join with Varying Workload

desired, the garbage collection time can be analytically removed from the reported blocking times using the same technique (see Section 3.7) that is used for removing the effects of crash timeout intervals in the correlated fault experiments.

4.1.4 Single Node Join

Experiments with single node join injections and varying message rate were conducted with a workload varying between 5 messages/sec to 500 messages/sec, and a fixed group size of 4 nodes before the node join. Hence, 5 nodes participated in the group membership protocol. The results for these experiments are shown in Figure 4.4. The results, as expected, do not show any of the flow control layer effects. As can be seen from the figure, the total blocking times for both the core and full protocols are fairly low (less than 8 msec) and do not increase appreciably as message rate was increased. However, the partial blocking time for the full protocol is quite large. This was unexpected, because the large difference between the partial and total blocking times means that either the preblock or postblock stagger are large, and hence the nodes spent a large portion of their time waiting on one (or more) of the other nodes. Moreover, the large difference between the partial blocking times for the full and core protocols means that it is the time spent waiting for all the nodes to begin the state transfer protocol that is large. Such behavior was not observed with the results for crash failures. An examination of the global timelines revealed the reason for the waiting. It was due to nodes from the primary group waiting for the core group membership protocol to

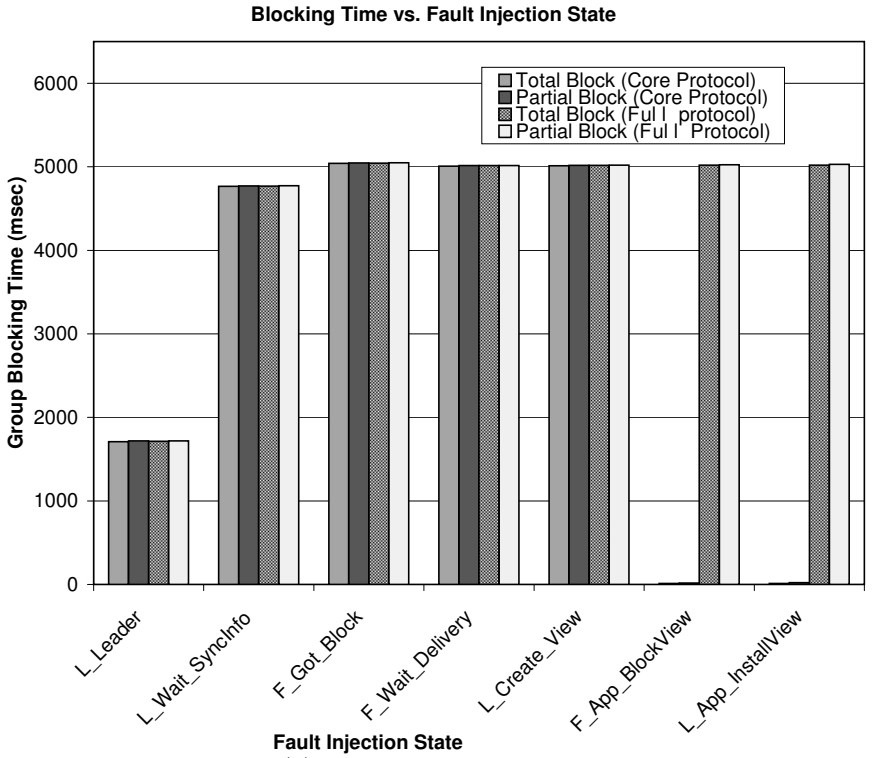
finish on the joining node so that they could all start executing the state transfer protocol. The blocking time at the joining node was not considered during computation of the core protocol, but it was considered for the state transfer part of the protocol, which led to the large difference.

The experiments for node join injections with varying group size were conducted with a fixed workload of 10 messages/sec, and a group size varying between 2 and 9 nodes before the node join. Hence between 3 and 10 nodes participated in the group membership protocol. These experiments yielded total blocking times that grew fairly slowly. However, the partial blocking times grew much more quickly than the total blocking times. This was expected, because synchronization delays are expected to increase with increasing group size.

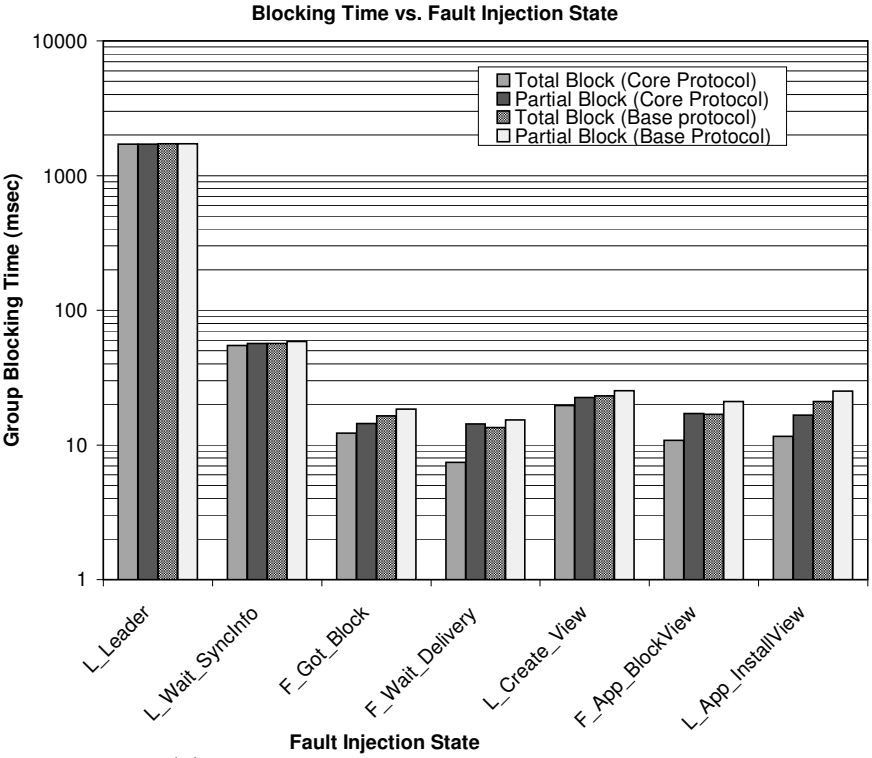
4.2 Correlated-Failure Experiments

The correlated fault experiments inject two crash failures into the group: one during normal operation, and a second one in specific global states of the group membership protocol (specified in Section 3.6.3) while it is trying to tolerate the first crash failure. These experiments are designed to measure the robustness of the fault tolerance mechanism to additional faults. The initial group size was fixed at 5 nodes, and the message rate was set to 10 messages/sec. The crash detection timeout was set to 5 seconds. Figure 4.5(a) presents the resulting group blocking times. The labels on the X axis denote the state in which the second fault was injected (as described in Section 3.6.3). It can be seen that for most of the experiments, blocking times were dominated by the crash detection timeout, with three exceptions. The blocking times for the core protocol were low when faults were injected into the `F_App_BlockView` and `L_App_InstallView` states, which was natural because the definition of the core protocol does not include the state transfer protocol of which these states are a part. Using the same argument, injection of a crash in the `Leader` state to prevent the leader from sending a *Block* message should have resulted in blocking times identical to those in the single fault experiments. The only change should have been a doubling of the crash detection timeout period. However, this was not the case.

To investigate this phenomenon and also to gain insight into the effect of correlated crashes on other states, the effects of the timeout were removed as described in Section 3.7, thus representing an idealized scenario of instant crash detection. The resulting group blocking times are shown in Figure 4.5(b). Comparing these times to the group blocking times for a single crash with a workload of 10 messages/sec (Figure 4.2(a)), modest increases between 10% to 100% can be seen for most of the states in the case of correlated injections.



(a) Actual Blocking Times



(b) Blocking Times with timeout factored out

Figure 4.5: Correlated Crash-failure Injections for a Stack Without Flow Control

The two exceptions are the `Leader` state and the `L_Wait_SyncInfo` state, for which blocking times are unusually high. An examination of the global timelines revealed that the reason is the sequencer-based total ordering protocol. The protocol works by having followers unicast broadcast messages to the leader, which then sequences and broadcasts the messages to the group. When the leader crashes before the group is blocked, any subsequent broadcast messages sent to the leader are not delivered and are buffered by their senders. When the members learn of the leader failure, they broadcast the buffered messages to the rest of the group. The burst transmission of accumulated messages from all members of the group causes the membership protocol to take longer than usual to complete. This phenomenon is the cause of the high blocking times for correlated injections in the `Leader` state and the `L_Wait_SyncInfo` state. The problem does not arise when the leader crashes after the group has blocked, because new message transmissions cannot be generated in a blocked group. However, the problem is easy to solve by explicitly notifying the application to stop sending broadcast messages when the unicast flow control layer runs out of credits. Since the new Ensemble application interface has support for explicit flow control notification, this modification should be trivial to implement.

4.3 Commentary on Results

The results of the experiments highlight how the complex interactions of flow control with the group membership protocol can have a dramatic effect on the availability of the system. The reduction in availability due to leader failures cannot be less than the crash detection timeout period. However, it can be limited to the timeout period if explicit application notification is used to limit broadcast-message traffic to the leader, when it fails to acknowledge a certain number of unicast messages. However, for single-follower failures, the situation is more complex. Buffering outgoing messages and not sending them to any group member during the timeout interval can be devastating from an availability standpoint, as evident from the results. The new application interface in Ensemble offers the facility of explicitly notifying the application to stop transmission; that facility alleviates the problem to some degree, but causes the timeout interval to be the bottleneck on system availability every time a crash occurs. The alternative scheme, of permitting outgoing message delivery during timeout periods, increases the memory requirements for unacknowledged message buffers. However, this scheme can reduce blocking time for crash failures substantially as evidenced by the results, and hence it seems to be favorable from an availability standpoint.

Chapter 5

Conclusion

This thesis has presented work on global-state-based fault injection and its use in evaluating the group membership protocol of the popular Ensemble group communication system. This study is one of the first times the technique of global-state-based fault injection has been used for the experimental evaluation of a distributed system implementation. Consequently, our work began by extending the Loki fault injector, which is unique in its ability to perform such fault injection. This thesis described the development of the measures language component of Loki. Important contributions of this thesis to the technical foundations of global-state-based fault injection include development of algorithms for the problem identifying incorrect fault injections, and the proof of the NP-Completeness (in the size of the fault trigger) of the problem. We also described extensions to the Loki measures language that were done to improve its expressive power and flexibility. The thesis also contained a performance evaluation of the Loki tool. This evaluation attempted to quantify properties such as precision, intrusiveness, and accuracy of the tool. These properties will help users understand the strengths and limitations of both the technique and the tool, and should prove to be a valuable reference for future Loki users.

After describing our work on the Loki fault injector, this thesis described a case study of the group membership component of the Ensemble GCS, and the results obtained from fault injecting it. The case study made two important contributions. First, being one of the first case studies involving the technique of global-state-based fault injection, it provided a proof-of-concept for this technique, and will serve as a reference for its future use. The case study described a novel technique for the generation of state-machine descriptions of the protocol using event-chains. There is reason to believe that the technique has wider applicability, and can can be used on other kinds of systems as well. The case study also demonstrated two slightly different uses of global-state-based fault triggers and global timelines. We showed how global-state-based fault triggers can be used for their intended purpose to inject faults

based on the state of multiple nodes in the system. However, they can also be used to temporally or spatially coordinate the injection of multiple faults in an elegant manner. The global timelines can be used to compute system measures, but they can also be used as traces for examining the behavior of the distributed system. We showed that such an examination can be very useful in debugging a distributed system.

The second contribution made by the case study was from the results of fault injection on the Ensemble group membership protocol. These results quantify the blocking behavior of the group membership protocol in response to single as well as multiple group membership events. This data can be used by designers of applications written on top of Ensemble to quantify the impact of the group membership protocol on the availability of their applications. More importantly, the results demonstrate that credit-based flow control can interfere with the virtual synchrony components of the group membership protocol and seriously affect the availability of the system. We made suggestions of ways to help mitigate this problem. Briefly, for applications with stringent availability requirements, a rate-based flow control scheme may be a better solution that credit-based flow control. Based on the results of fault injection, we also suggested other changes to the protocol that could potentially help improve its performance and availability characteristics under certain situations. The insight gained from our study will be useful to designers of group membership systems.

5.1 Future Work

The safety and performability demands being placed on distributed systems are constantly increasing as society becomes increasing dependent on those systems. However, such distributed systems are also becoming much larger and more complex, thus giving rise to emergent behavior, and making them harder to evaluate and verify using modelling techniques. Thus, we believe that further work on experimental validation of distributed systems would have great value.

The work described in this thesis has extended the experimental evaluation technique of fault injection to the domain of distributed systems that rely on a notion of global state, and has shown that the technique can expose emergent behavior that may result from unexpected interactions between the components of a complex system. However, this thesis has only scratched the surface, and much work remains to be done in this area. Some promising areas for further exploration are outlined below.

5.1.1 Fault and Trigger Template Library

The faults used in this thesis were very simple. They included crash failures and node joins. However, since Loki does not impose restrictions on the fault model, it can be used to inject a wide array of faults. While many of these faults may be application-specific, there is reason to believe that several types of faults may be common to many distributed systems. As described in Chapter 1, faults that have already been studied in the literature include crash failures, network partitions, message loss, corruption or reordering, bit-flips, and illegal branches. Other kinds of faults, such as slowing down of processes (to test synchrony assumptions of a protocol), Byzantine faults, and timing errors, have not yet been explicitly used in the context of fault injection, but are applicable for a wide variety of systems. It would thus greatly enhance the ease of use of Loki (or any other fault injector) if these common distributed system fault models were implemented in the form of a fault library. The associated research issues involve identifying fault models common across a range of applications and parameterizing them such that they are both easy to use and widely applicable. Similarly, several types of global state triggers are applicable to situations that may occur frequently in distributed systems. Although the exact fault trigger would vary with the state machine definition, the general patterns would be retained. An example in this thesis is the general templates used for injections of a fault when a leader or follower is in a particular set of states. That situation would appear in many leader-based distributed protocols. Identification of such common situations and generation of templates for them would provide practitioners with a useful resource, and greatly increase the ease of use of the global-state-based fault injection technique.

5.1.2 Model-Driven Fault Injection

The use of a state machine abstraction to drive fault experimentation and fault injection has been an important part of this thesis. Using the observation that a state machine is actually a *model* of the system under study, the state-based fault injection approach can be generalized into a more generic *model-driven fault injection* technique. In this technique, the system under study can be modelled using any of the modelling formalisms for dependability considered in the literature. These include formalisms based on Petri nets (such as Stochastic Activity Networks [SM01] or Generalized Stochastic Petri Nets [MBC+95]) or formalisms based on queueing networks, such as [Hav95]. The system would still be instrumented using events, but the events would be generalized in that they would convey information appropriate to the formalism in which the model was represented (for example, they could describe token changes if Petri nets were used). Using such formalisms to drive fault injection would

serve three equally important purposes. First, the application state would be more compactly represented, and much more complex semantics could be handled. Second, the model could also act as a generalized fault trigger, and help drive the fault injection process. This would be useful because modelling formalisms such as SANs are Turing-complete. Hence, it would be possible to perform fault injections based on conditions far more complex than could be used with simple Boolean triggers. Third, the models could be extensions of the same models that were used to validate the system during its design phase, and could use the same measures. That would provide a uniform platform for system evaluation during both the early design phases and the later implementation phases of a project.

5.1.3 Intrusion Injection

With the emphasis being placed on the survivability of critical systems and infrastructures on the rise, intrusions are being investigated as an important class of faults. However, intrusions differ from classical faults in that the fault generating mechanisms (i.e., human intruders) follow complex processes that may depend on environmental factors and observations of the system under attack. If intrusions are viewed as faults, then in fault injection terminology, the previous statement means that the fault triggers are complex. If the system under consideration is a distributed system, the fault triggers may depend on its global state because attackers may utilize all the information they have about the different nodes of the system while perpetrating an attack. Previous work in the area of modelling intrusion tolerance suggests [JO97] that attack behavior can be modelled using Markov chains. More recent work [SCS03] uses information about the global state of the system (the state of multiple servers) to determine future attack rates. Extended to support model-driven injection as described above, global-state-based fault injection could thus be very useful in quantifying the effect of intrusions on distributed system implementations, and for experimentally validating intrusion tolerant systems.

Further work in these areas will hopefully serve to emphasize the role of experimental techniques in the evaluation and validation of systems, expand the domain of such techniques, and add to the repertoire of techniques available to practitioners in this field.

References

[AAA+90] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martin, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, Feb 1990.

[AAC+90] 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*, 39:455–467, Oct 1990.

[Bir85] K. P. Birman. Replication and fault tolerance in the ISIS system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 79–86, Orcas Island, Washington, Dec 1985.

[Bir96] K. P. Birman. *Building Secure and Reliable Network Applications.* Manning, 1996.

[BJS+95] D. Bhatt, R. Jha, T. Steeves, R. Bhatt, and D. Wills. SPI: An instrumentation development environment for parallel/distributed systems. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 494–501, 1995.

[CCH+99] M. Cukier, R. Chandra, D. Henke, J. Pistole, and W. H. Sanders. Fault injection based on the partial global state of a distributed system. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 168–177, Oct 1999.

[CCLS00] 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*, pages 248–257, Oct 2000.

[Cha00] R. Chandra. Loki: A state-driven fault injector for distributed systems. Master's thesis, University of Illinois at Urbana-Champaign, 2000.

[CLCS00] 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)*, pages 237–242, Jun 2000.

[CLJ+02] R. Chandra, R. M. Lefever, K. Joshi, M. Cukier, and W. H. Sanders. A global-state-triggered fault injector for distributed system evaluation. Submitted for publication, 2002.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, MIT Press, New York, 1990.

[CPA99] M. Cukier, D. Powell, and J. Arlat. Coverage estimation methods for stratified fault-injection. *IEEE Transactions on Computers*, 48(7):707–723, Jul 1999.

[CT96] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar 1996.

[DJ95] S. Dawson and F. Jahanian. Probing and fault injection of dependable distributed protocols. *The Computer Journal*, 38(4):286–300, 1995.

[DJMT96] S. Dawson, F. Jahanian, T. Mitton, and T. L. Tung. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, pages 404–414, Jun 1996.

[EK73] C. E. Ellington and R. J. Kulpinski. Dissemination of system time. *IEEE Transactions on Communications*, COM-21:605–623, May 1973.

[EL92] K. Echtle and M. Leu. The EFA fault injector for fault-tolerant distributed system testing. In *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 28–35, 1992.

[FLS01] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, 2001.

[Hav95] B. R. Haverkort. Performability evaluation of fault-tolerant computer systems using DyQN-Tool+. *International Journal of Reliability, Quality, and Safety Engineering*, 2(4):383–404, 1995.

[Hay97a] M. Hayden. *Ensemble Reference Manual*. Cornell University, 1997.

[Hay97b] M. Hayden. *Ensemble Tutorial*. Cornell University, 1997.

[Hay97c] M. G. Hayden. *The Ensemble System*. PhD thesis, Dept. of Computer Science, Cornell University, Jan 1997.

[Hen98] D. A. Henke. Loki – An empirical evaluation tool for distributed systems: The experiment analysis framework. Master's thesis, University of Illinois at Urbana-Champaign, 1998.

[HLvR99] J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for ensemble layers. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 119–133, Mar 1999.

[HSR95] S. Han, K. G. Shin, and H. A. Rosenberg. DOCTOR: An integrated software fault injection environment for distributed real-time systems. In *Proceedings of the International Computer Performance and Dependability Symposium*, pages 204–213, 1995.

[JCS02] 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*, pages 140–158, Toulouse, France, October 2002.

[JO97] E. Jonsson and T. Olovsson. A Quantitative Model of the Security Intrusion Process Based on Attacker Behavior. *IEEE Transactions on Software Engineering*, 23(4):235–245, April 1997.

[KHH98] C. Kreitz, M. Hayden, and J. Hickey. A proof environment for the development of group communication systems. In *Proceedings of Automated Deduction - CADE-15 15th International Conference on Automated Deduction.*, pages 317–332, Lindau, Germany, Jul 1998.

[Lap95] J. C. Laprie. Dependable computing: Concepts, limits, challenges. In *25th IEEE International Symposium on Fault-Tolerant Computing - Special Issue*, pages 42–54, 1995.

[Lef03] R. M. Lefever. An experimental evaluation of the coda distributed file system using the loki state-driven fault injector. Master's thesis, University of Illinois at Urbana-Champaign, 2003.

[LKG92] F. Lange, R. Kroeger, and M. Gergeleit. JEWEL: Design and implementation of a distributed measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 3:657–671, Nov 1992.

[MBC+95] M. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley and Sons, Chichester, England, 1995.

[ML98] S. Mishra and W. Lei. An evaluation of flow control in group communication. *IEEE/ACM Transactions on Networking*, 6(5):571–587, Oct 1998.

[Nei96] G. Neiger. A new look at membership services. In *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, pages 331–340. ACM Press, May 1996.

[SCS03] S. Singh, M. Cukier, and W. H. Sanders. Probabilistic validation of an intrusion-tolerant replication system. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2003)*, page To appear, San Francisco, CA, Jun 2003.

[SFB+00] D. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. Iyer. NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS 2000)*, pages 91–100, 2000.

[SHRI97] D. Stott, M. Hsueh, G. Ries, and R. Iyer. Dependability analysis of a commercial high-speed network. In *Proceedings of the Symposium on Fault-Tolerant Computing*, pages 248–257, 1997.

[SM01] W. H. Sanders and J. F. Meyer. Stochastic activity networks: Formal definitions and concepts. *Lectures on Formal Methods and Performance Analysis, First EEF/Euro Summer School on Trends in Computer Science, Berg en Dal, The Netherlands, July 3-7, 2000, Revised Lectures, Lecture Notes in Computer Science no. 2090*, pages 315–343, 2001.

[SSK+01] D. Stott, N. Speirs, Z. Kalbarczyk, S. Bagchi, J. Xu, and R. K. Iyer. Comparing fail-silence provided by process duplication versus internal error detection for DHCP server. In *Proceedings of International Parallel and Distributed Processing Symposium, IPDPS'01*, Apr 2001.

[UMK97] P. W. Uminski, M. R. Matuszek, and H. Krawczyk. Experimental evaluation of PVM group communication. In *Proceedings of Recent Advances in Parallel Virtual Machine and Message Passing Interface. 4th European PVM/MPI Users' Group Meeting.*, pages 57–63, 1997.

[VKCD99] R. Vitenberg, I. Keidar, G. Chockler, and D. Dolev. Group communication specifications: A comprehensive study. Technical report, Tech. report CS99-31, Comp. Sci. Inst., The Hebrew University of Jerusalem and MIT Technical Report MIT-LCS-TR-790, Sep 1999.

[vRBH+98] Robbert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburd, and David A. Karr. Building adaptive systems using Ensemble. *Software - Practice and Experience*, 28(9):963–979, 1998.

Appendix A

Syntax for Loki Measures

Notation	Meaning
Sm	State machine name
St	State name
Ev	Local event name
t, tLo, tHi	Instant of time in microseconds (experiment begins at time 0)
Guard, Expr	Expression using C expression syntax
$Name_1$	Any string (usually a state machine, state, event or fault name)

Category	Syntax
Functions/Definitions usable throughout entire tuple	
chk_fault	loki_chk_fault1(Sm, Fault, Label) Returns TRUE if the given fault on state machine Sm was labelled with the given Label.
	loki_chk_fault2({$Name_1, \ldots$}, Fault, Label) Returns the number of state machines from the list on which the given fault had the given label.
Label	CORRECT|INCORRECT|NO_INJECTION|INJECTION The INJECTION type is used when it does not matter whether the fault was injected correctly or not.
Time	EXP_START_TIME Returns the time at which the experiment began (usually 0).
	EXP_END_TIME Returns the time at which the experiment ended.
Observations	LOKI_OBS_VALUE Refers to the observation value of the current tuple.

Category	Syntax
Functions/Definitions usable in Predicate Functions	
chk_state	loki_chk_state1(Sm, St) Returns TRUE if Sm is in state St.
	loki_chk_state2(Sm, St, t) Returns TRUE if Sm is in St at time t.
	loki_chk_state3(Sm, St, tLo, tHi) Returns TRUE if Sm is in St and the current time is between tLo and tHi.
chk_event	loki_chk_event1(Sm, St, Ev) Returns TRUE if event Ev was generated at Sm when it was in St.
	loki_chk_event2(Sm, St, Ev, tLo, tHi) Returns TRUE if Ev was generated at Sm when it was in St, and the current time is between tLo and tHi (both inclusive).
	loki_chk_event3(Sm, Ev) Returns TRUE if Ev was generated at Sm.
	loki_chk_event4(Sm, Ev, tLo, tHi) Returns TRUE if Ev was generated at Sm and the current time is between tLo and tHi.
for_all	loki_for_all1(ListType, [x, Guard], [x, Expr]) Returns TRUE if all members in ListType that pass Guard return TRUE for Expr.
	loki_for_all2({$\text{Name}_1, \ldots$}, [x, Guard], [x, Expr]) Returns TRUE if all members in explicit list that pass Guard return TRUE for Expr.
if_any	loki_if_any1(ListType, [x, Guard], [x, Expr]) Returns TRUE if there is any member in ListType that passes Guard and for which Expr returns TRUE.
	loki_if_any2({$\text{Name}_1, \ldots$}, [x, Guard], [x, Expr]) Returns TRUE if there is any member in the explicit list that passes Guard and for which Expr returns TRUE.
how_many	loki_how_many1(ListType, [x, Guard], [x, Expr]) Returns the number of members in ListType that pass Guard for which Expr is TRUE.
	loki_how_many1({$\text{Name}_1, \ldots$}, [x, Guard], [x, Expr]) Returns the number of members in the explicit list that pass Guard for which Expr is TRUE.
ListType	SMLIST Selects list of all State Machines in the system.
	EVENTLIST Selects list of all Events in the system.
	STATELIST Selects list of all States in the system.
	FAULTLIST Selects list of all Faults in the system.

Category	Syntax
Functions/Definitions usable in Observation Functions	
	count(Level, Transition, tLo, tHi) Returns the number of times the predicate timeline undergoes the given transition between time tLo and tHi, starting from the given Level.
	outcome(t) Returns the truth value of the predicate timeline at time t
	duration(Truth, x, tLo, tHi) Returns the time for which the predicate timeline has the given truth value for the x^{th} time in the interval between tLo and tHi.
	instant(Level, Transition, x, tLo, tHi) Returns the instant at which the predicate timeline attains the given truth value for the x^{th} time between the time tLo and tHi.
	total_duration(Truth, tLo, tHi) Returns the total amount of time for which the predicate timeline has the given truth value between times tLo and tHi.
Truth	TRUE|FALSE
Level	UP|DOWN|BOTH
Transition	IMPULSE|STEP|ALL The IMPULSE type is used for events.

Appendix B

OCAML-Loki Interface

The OCAML-Loki interface allows an application written in OCAML to be interfaced with the Loki runtime, pass local event notifications to Loki, and be injected with faults. The OCAML interface is contained in the library `mlloki.cma` in the Loki distribution. To attach the runtime to the OCAML application, one simply compiles the application with this library. No other instrumentation is needed. With the runtime attached, the following methods are available to the application.

Method	**Signature**	**Description**
`notifyevent`	*string* → *unit*	Notify the Loki runtime of an event with the given name. The time associated with the event is the time of the function call.
`notifyeventat`	*string* → *time* → *unit*	Notify the Loki runtime of an event with the given name and occurrence time. time is an opaque structure that cannot be manipulated by the OCAML program.
`getcurrenttime`	*unit* → *time*	Returns an opaque time structure that contains the time of the function call.
`timetostring`	*unit* → *string*	Converts a time structure to a string for display and debugging purposes.

The fault injection is done by writing a C `mlinjectFault` function to implement the fault, and linking the object file for this function with the OCAML application. Alternatively, fault routines can be implemented in OCAML by implementing a `loadClosures`:*unit* → *unit* function in OCAML that accepts and returns no parameters, but implements a closure named `injectFault`:*string* → *unit* that accepts a fault name. If present, this closure will be called by the Loki runtime with the fault name to be injected.

Appendix C

Ensemble Event Flow

This chapter shows the event flows for the Ensemble group membership protocol. These event flows were generated manually by inspecting the source code. The event chains for the membership protocol and the state machines for fault injection were derived using these event flows. All events occurring on a leader are colored Red. All events occurring on a follower are colored blue. The black circles indicate times when the protocol waits for communication from another node.

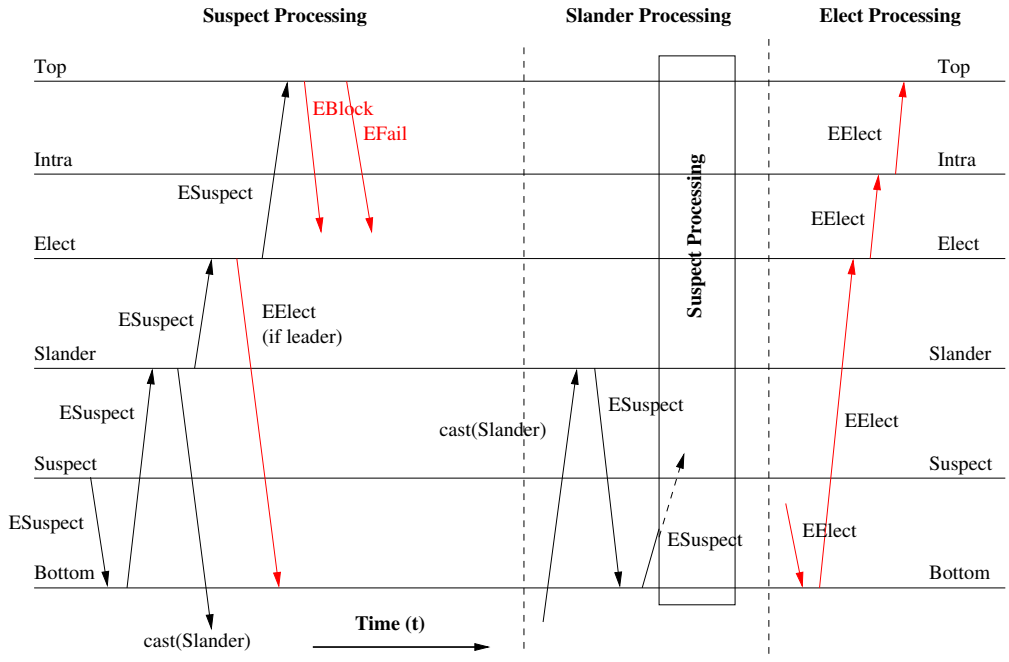


Figure C.1: Event Flow for Crash Failure Detection (Detection Phase)

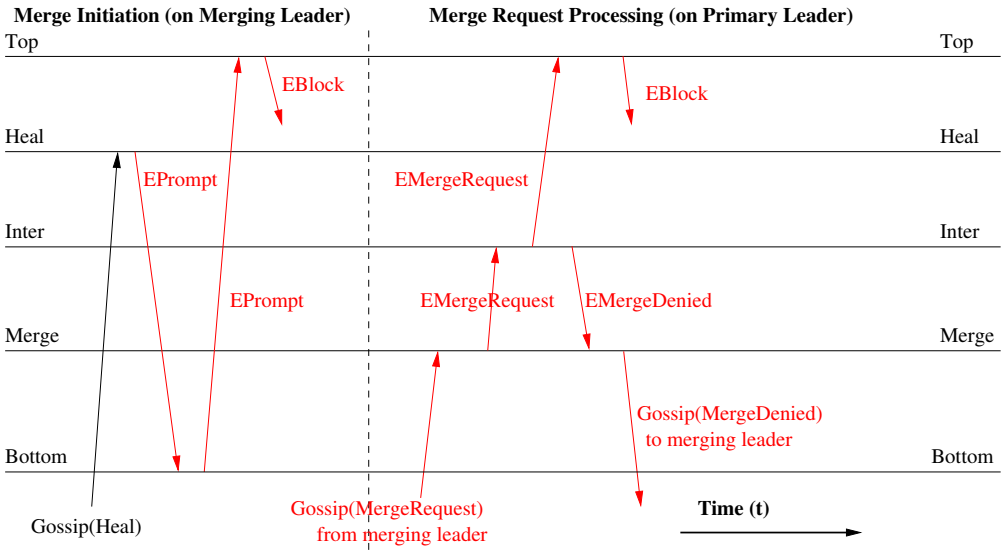


Figure C.2: Event Flow for Merge Initiation and Response (Detection Phase)

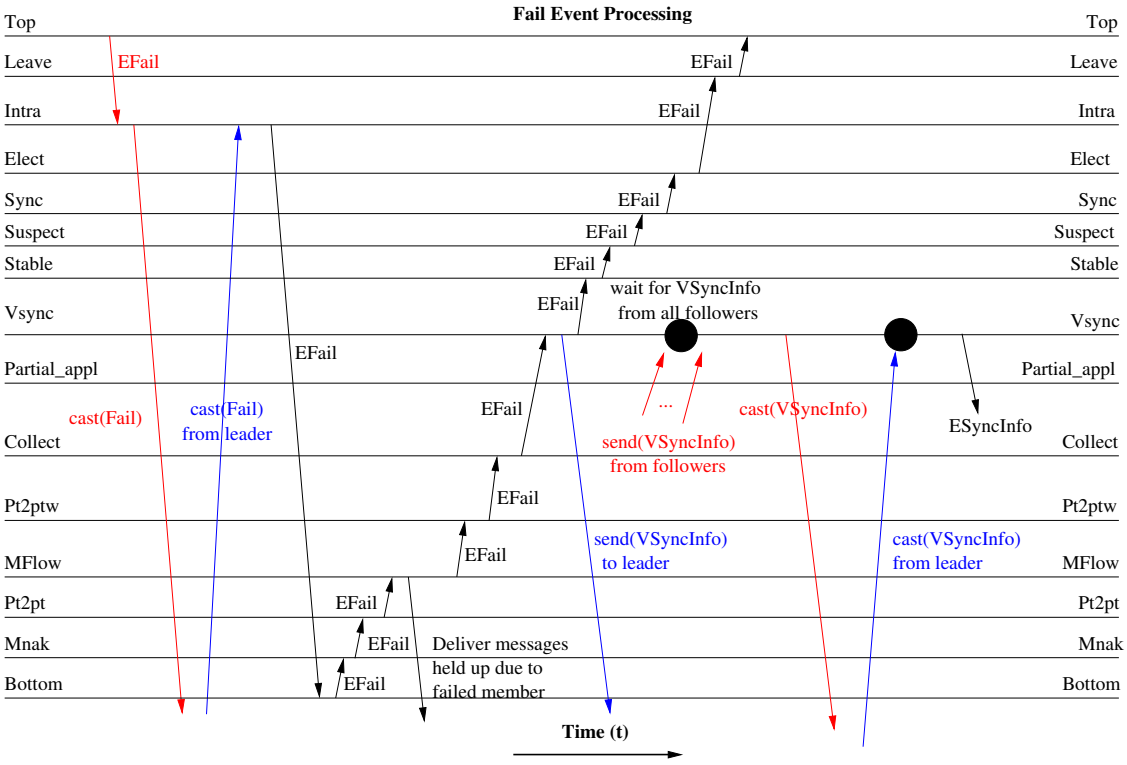


Figure C.3: Event Flow for Failure Event Handling (All Phases)

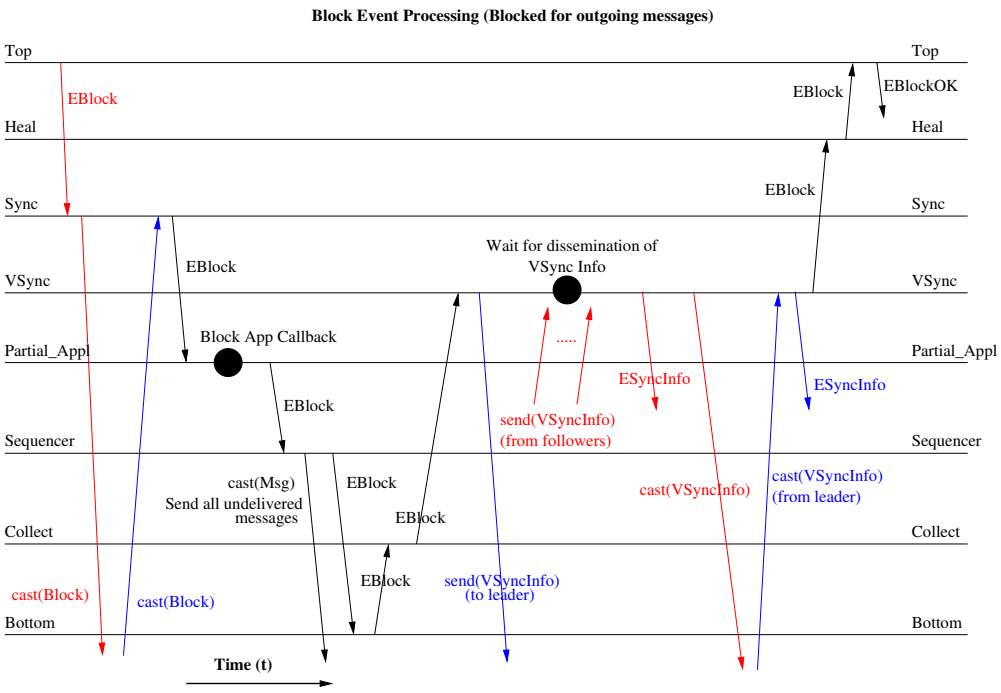


Figure C.4: Event Flow for Block Event (Prepare Phase)

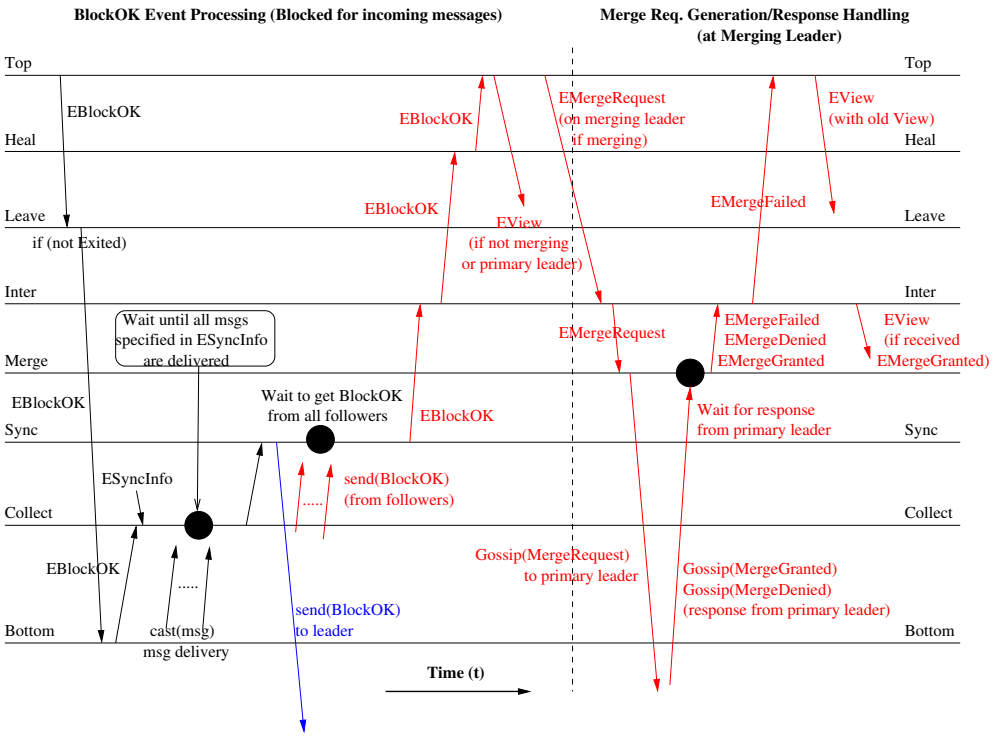


Figure C.5: Event Flow for BlockOK and Merge Events (Delivery and Merge Phases)

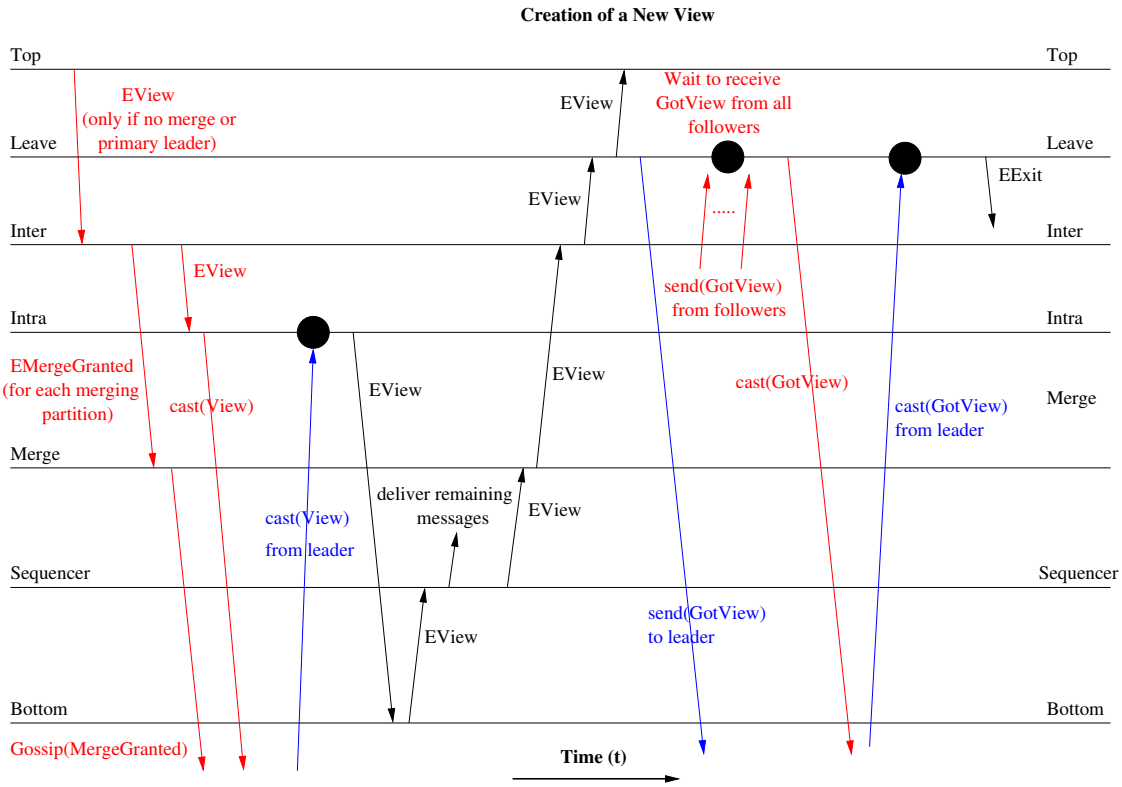


Figure C.6: Event Flow for New View Creation (Create View Phase)