LOKI--AN EMPIRICAL EVALUATION TOOL FOR DISTRIBUTED SYSTEMS: THE EXPERIMENT ANALYSIS FRAMEWORK

BY

DAVID ALAN HENKE

B.S., Michigan Technological University, 1996

THESIS

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

Urbana, Illinois

To my parents

ACKNOWLEDGEMENTS

There are many people whom I wish to thank for their help and support. Dr. William Sanders, my advisor, guided and supported me not only in the work reported in this thesis, but in all aspects of my graduate education at the University of Illinois. Dr. Michel Cukier helped with the design and development of Loki in general, and the implementation of clock synchronization in particular, and provided many valuable comments on earlier revisions of this thesis. Jessica Pistole, as developer of the Loki run time, was integral to the development of Loki and provided me with many helpful suggestions and insights in my work. Alex Williamson assisted in the maintenance of the testbed on which Loki was developed. Jenny Applequist aided in the editing of this thesis and presentations regarding Loki. I would also like to thank all of the members of the Performability Engineering Research Group, especially Chetan Sabnis, and contacts at BBN Technologies for their general support and advice. Finally, I want to thank the Defense Advanced Research Projects Agency Information Technology Office, which funded the work reported in this thesis under contract F30602-96-C-0315.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

1. INTRODUCTION

1.1. Dependability Evaluation

Dependability is an important attribute of all computing systems. However, the necessary level of dependability varies from system to system. Whatever the required level, users need confidence that it is provided by the system. The validation of the system, via both verification and evaluation, can provide that confidence [1].

Evaluation of a fault-tolerant system involves the determination of the efficiency of its fault-tolerant mechanisms in covering the faults and errors (the activation of a fault) for which they were designed. Before one can properly evaluate a system, a thorough understanding of those mechanisms is necessary. Through the use of modeling, a proper insight into system operation can be gained. However, parameters related to fault-tolerant mechanisms are particularly difficult to obtain by modeling.

Dependability evaluation can therefore be accomplished through the integrated use of modeling and fault injection [2]. Fault injection can be used not only for evaluation (fault forecasting), but also for verification (fault removal). Fault injection allows, by verification, the identification and removal of faults in fault-tolerant implementations. In addition, fault injection also allows evaluation of the efficiency of the operational behavior of fault-tolerant mechanisms (fault forecasting). The operational behavior of fault-tolerant mechanisms is characterized in terms of measures such as coverage factors, dormancy, and latency. Coverage is the probability of system recovery given that a fault exists [3]. Dormancy and latency are times rather than probabilities. Dormancy is the time corresponding to the activation of an injected fault as an error. Latency is the time from the error to either failure of the system or detection and system recovery [2].

Coverage estimations of fault-tolerant mechanisms can be obtained from fault-injection experiments. To relate these conditional measures to the overall system dependability, factors such as fault occurrence and activation rates must be known. By integrating conditional measures obtained through fault-injection experimentation with system models of fault occurrence and activation rates, measures of the overall system dependability

can be obtained. For more details about the combination of fault injection and modeling for system evaluation, see [2]. The Loki empirical evaluation tool has been designed to facilitate the fault-injection experimentation necessary for proper system evaluation, while providing a framework general enough to be also used for fault removal.

1.2. Considered Systems

The use of specialized hardware is needed in order to guarantee that systems that need high dependability meet their dependability requirements. However, systems that have moderate dependability demands can use commercial off-the-shelf (COTS) systems and software-implemented fault tolerance to provide the needed properties. In particular, distributed systems composed of COTS components can provide the dependability desired through hardware replication and the use of such software services as group membership protocols, atomic multicast, and virtual synchrony. With the right tool support, fault injection can be used to evaluate the dependability of reliable distributed systems built from COTS components, as well as systems built with specialized hardware.

1.3. Existing Tools

Many tools have been written for the purpose of observing and testing distributed systems. Some of them, such as JEWEL [4], provide only measurement capabilities. JEWEL provides a global framework for system observation and measurement based on user-defined measures. Results of such observations can be analyzed and displayed in a real-time fashion. Such tools can observe a system under study, but cannot control the environment necessary to evaluate fault-tolerant systems (e.g., by injecting faults).

Other tools, such as Orchestra [5], DOCTOR [6], and EFA [7], provide fault-injection capabilities. Each of these tools provides the capability of injecting faults into the system under study based on the state of the local process they are monitoring. DOCTOR, a tool designed for injection of hardware faults, is implemented for the HARTS real-time operating system. EFA provides fault injection, but only in the data link layer. Orchestra is similar to EFA, but allows for the injection of faults at any level of the system desired. However, none of these tools provides the ability to inject faults in a local process based on the global state of a distributed system.

Another fault-injection tool, CESIUM [8], provides the ability to inject faults based on a global understanding of the distributed system. This is done by running the system under study in a simulated environment. However, for proper evaluation of a system, it is desirable to experiment on a real system since the operation of a system under study in a simulated environment cannot fully mirror its operation in a real environment.

SPI [9], from Honeywell, also allows for user-specified observation and validation of systems based on a global system view. This is done through the use of "ea-machines" that observe events in the system and execute "actions." The machines communicate amongst themselves to maintain a global understanding of the system. This system was designed with measurement and evaluation in mind. While it was not designed with fault injection in mind, SPI could also be used for fault injection. However, SPI lacks the features necessary to check for proper fault injection, synchronize clocks, and generate conditional measures. The SPI system currently runs on SUN workstations and Intel Paragon, but is being ported to other platforms.

1.4. Research Objectives

While the tools described in the previous section have been successfully applied to many systems, they lack two important capabilities for evaluating reliable distributed systems. Specifically, they lack the ability to inject faults based on the global system state in a real environment and the ability to provide statistically significant estimations, such as coverage and performance-related measures, based on experiment observations. The goal of this thesis is to design an empirical evaluation tool, called Loki, that has these capabilities, and to implement the experiment analysis capabilities of the tool. The run-time portion of the tool is described in [10].

To achieve the first capability, a mechanism is necessary to determine the global state, to provide a means to inject faults while in that particular system state, and to test after the experiment is done to verify that the fault was injected in the proper state. These mechanisms are necessary both for evaluation of a system given the occurrence of faults in specific global system states and for fault removal.

Loki provides the capability to evaluate a system based on global state through the use of state machines [10]. For a given experiment, a user defines the faults to be injected, the

state machines that maintain necessary information about the system, and the measures to observe. Different user-specified experiments are run and, based on communication between state machines, faults can be injected and observations made based on global state.

The observations from fault-injection experiments must be interpreted to produce statistically significant results concerning coverage and performance-related measures. Loki provides user-specified conditional measures that can be used for determination of overall measures of system dependability.

The implementation of facilities necessary to provide accurate results based on the observations made during fault-injection experiments is the focus of this thesis. First, synchronization of clock readings is necessary in order to provide a single global timeline on which all observations are based. This must be done in a manner that is not intrusive to the system under study, yet provides precise and accurate timestamps. Second, to ensure that faults are injected when desired, results must be checked for proper fault injection. This is done in post-analysis, so that experiments with improperly injected faults will not be included when calculating measures.

In particular, this thesis will describe:

- An overview of the Loki empirical evaluation tool, including the runtime framework and analysis tools.
- Determination of a data-gathering technique to efficiently provide information for clock synchronization without perturbing the system under study.
- Implementation of a clock synchronization algorithm that provides a single accurate global timeline with physical bounds for all recorded events.
- Implementation of an accurate and nonintrusive method to timestamp events locally.
- Implementation of a technique to test for proper injection of faults after fault-injection experiments have been conducted.

2. LOKI ARCHITECTURE OVERVIEW

2.1. Overview

This chapter provides a general overview of the Loki empirical evaluation tool. The goals of the Loki evaluation system architecture are described in Section 2.2. The Loki architecture can be logically separated into several pieces. The Loki run-time architecture is discussed in Section 2.3, which describes how Loki observes the system under study and injects faults. Upon completion of fault injection and observation of the system, the analysis tools described in Section 2.4 interpret the results to provide useful statistics about the observations. Finally, Section 2.5 describes the steps necessary to complete the successful evaluation of a system using Loki.

2.2. Goals

Loki is designed to provide targeted injection of faults in particular system states. Due to the distributed nature of the systems under study, the system state must be definable in terms of multiple system components. The architecture designed to meet this goal is described in Section 2.3 and is discussed in further detail in [10].

The second goal is statistically sound evaluation of the system. While many fault injector tools exist, few provide information about the statistical accuracy of the results. Loki is designed to provide statistically significant interpretation of fault-injection results. Such results include coverage and performance-related measures. Mechanisms to meet this goal are described in Section 2.4.

2.3. Run-Time Architecture

To satisfy the goals outlined in the previous section, it is necessary to introduce the concept of system state. Whether it is injecting faults or recording observations, the Loki run-time architecture acts based on the system state. The use of system state by Loki is described in Section 2.3.1. How Loki functions in terms of the two views of state, global and local, is described in Sections 2.3.2 and 2.3.3, respectively. The Loki run-time architecture can be seen in Figure 1. For more detailed information on this architecture, see [10].

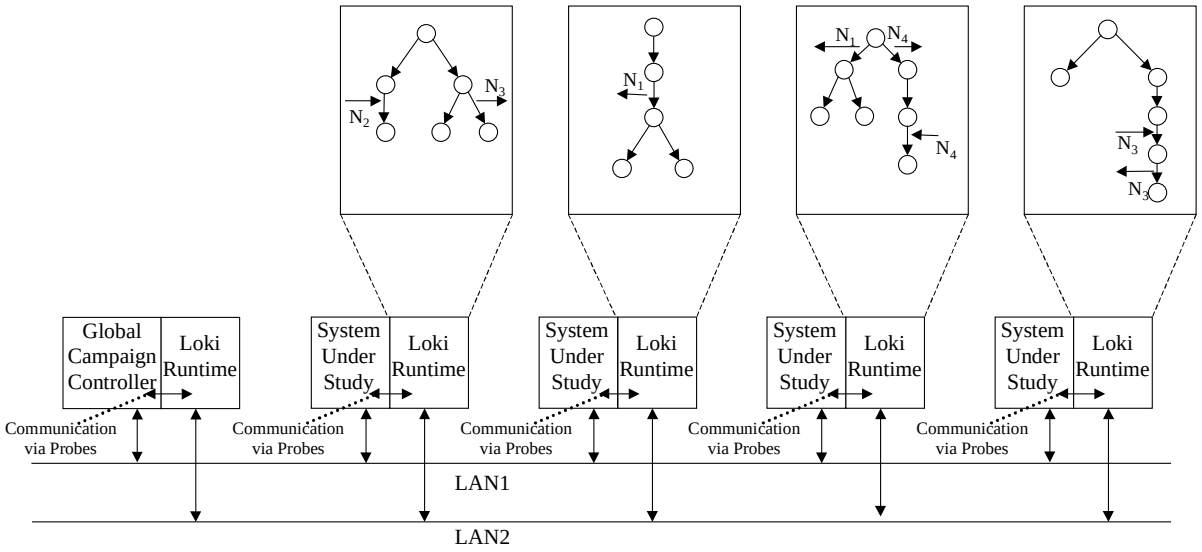

Figure 1: Global Architecture of Loki

2.3.1. System state

Each component of a distributed system is in some state. This "local" state applies only to the specific node. The combination of all local states of the system is the global state of the system. Normally, the local states in a distributed system are not shared among one another unless it is necessary. Thus, each node has only a "local view" of state. This is not sufficient when evaluating a distributed system or removing faults from the system, since the injection of faults into the system under study may be dependant on the state of more than just the local node.

To address this concern, there needs to be a mechanism to retain information regarding states of multiple nodes of the system. Loki could maintain a "global view" of the system. However, it is not necessary for an outside monitor to try to maintain detailed information about all of the local nodes of the system. In addition, obtaining and processing all of this information would be too intrusive to the system under study. A properly designed tool will maintain only the local state information from multiple states that is necessary to obtain measures and inject faults properly. This is referred to as a *measure-driven partial global view of state*. By maintaining this partial global view of the state, instead of just the view of local state that a node normally has, it is possible for Loki to direct specific observations and fault

injections. The necessary partial global view of state is driven by the measures desired; thus the term *measure-driven partial global view of state* is used to refer to the state kept by Loki.

The measure-driven partial global view of state is maintained by the Loki run-time architecture through the use of state machines. The distributed system under study is composed of several nodes. For each node that the user is interested in studying, there must be a state machine description of relevant states of that node. Relevant states include those in which faults are to be injected, those in which observations are to be recorded, and those in which events need to be observed if a correct view of system state is to be maintained.

2.3.2. Local architecture overview

The state machine gathers information about the local node through interfaces with the system under study called *probes*. These probes relay information back and forth between the state machine and the system under study. They can pass information regarding system state or observations to be made. They also can relay information, such as when to inject a fault, from the state machine.

The user must instrument the system under study to insert the probe. The probe and instrumented code may take the form of a function, some layer in a protocol stack, or an object wrapper. Whatever form it takes, it must provide the state machine with the ability to interact with the node when necessary.

One function of the probe is to provide the state machine with information regarding the system state. While the state of the state machine may be updated by notification messages from other nodes, it may also change due to events that occur in the local node. Information may be passed to the state machine to be recorded as observations, or to trigger notifications to be sent to other nodes.

Another function of probes is the injection of faults. Fault injections must be done in the instrumented code based on input from the probe. For that reason, the state machine must provide information on when it is appropriate to inject faults. As with any communication between the system under study and the state machine via the probe, information can be sent actively or passively from the state machine. An active state machine will send information as soon as it is available. A passive state machine will hold information until requested through the probe.

2.3.3. Global architecture overview

To observe the system and interact with it, a local state machine is associated with each node of the system under study. Each state machine keeps track of the state of the local node with which it is associated. By passing messages to one another when necessary, these local state machines can be updated on the status of other nodes of the system. The capability for a measure-driven partial global view of state is provided by these *communicating state machines*. There are also *noncommunicating state machines* that maintain only local state and do not have contact with other nodes. Noncommunicating state machines are less intrusive than communicating state machines and should be used whenever possible.

To prevent communicating state machines from being overly intrusive, the state machines are designed not to block when state update notifications are sent. This means that global views of the system may be inconsistent among the local state machines. Inconsistency may occur because a state machine will not wait to update its state until others have updated their states accordingly. To make sure that any inconsistencies do not affect the results, all faults injected are checked in postprocessing to guarantee that the local views of the system were consistent when the fault was injected. For more details about this fault-injection testing, see Section 2.4.2 and Chapter 5.

Since events such as observations, injections, and state changes are event-driven, no global clock is necessary during the experiment. When timing is necessary, local clocks can be used. All observations are recorded using local clocks. When analyzing the results, however, a single timeline of the experiment is necessary. For instance, a fault may be injected on one machine and covered on another. If a time to coverage is needed, then so is a single timeline. This is accomplished by adjusting the local timestamps of events in post-analysis. The synchronization of timestamps is explained further in Section 2.4.1 and in detail in Chapters 3 and 4.

Loki also has a global campaign controller. This campaign controller starts every experiment, automatically runs the appropriate number of experiments for the fault-injection campaign, and determines when a fault-injection campaign has finished.

2.4. Analysis Tools

After a set of experiments completes, the raw results must be processed and interpreted to obtain the desired measures. This includes synchronizing the recorded timestamps to a single timeline, testing faults to ensure proper injection, and estimation of the desired measures. This section provides an overview of the analysis tools. For more details about timestamp synchronization see Chapters 3 and 4. Fault-injection testing is discussed in detail in Chapter 5.

2.4.1. Timeline synchronization

Timeline synchronization is necessary for the interpretation of results after the experiment. Events during fault-injection campaigns are based on the state of the system or on local clocks. Events are not based on a global clock. Thus, a synchronized global clock is not necessary during fault-injection campaigns. During an experiment, recorded events are timestamped using a local clock. Once a set of experiments is concluded, the recorded observation times are adjusted to a single timeline. The timeline corrections are based on the assumption of linear clock drift. Due to uncertainty in synchronizing the local readings, physical maximum and minimum bounds are calculated for each timestamp with respect to the global timeline.

2.4.2. Fault-injection testing

During each experiment, faults are injected based on what a local node views as the global state of the system. Due to delays when passing messages in the system, some local nodes may update their global view of state before others. This can lead to inconsistent views of global states across the nodes. If a fault is injected when the views of global state are inconsistent, the injection of the fault at the desired time cannot be guaranteed.

To avoid that problem, the fault injections are tested after the experiment is done. If the global views of state are inconsistent at the time of injection, the fault-injection experiment results are thrown out. Proper fault injection may depend on only some of the nodes agreeing on the global state of the system. To limit the fault-injection testing to only those nodes, *fault conditionals* are defined by the user. In that way, only the necessary nodes

are tested for agreement on the time and state where the fault was injected. Fault-injection testing is discussed in detail in Chapter 5.

2.4.3. Measure estimation

After the timestamps are synchronized and faults are tested for proper injection, the desired measurements from the experiment can be obtained. The measure estimation capabilities of Loki allow for the determination of coverage and performance-related measures. Specific measures that might be desired by a user include the probability that a fault is covered, the time from fault occurrence to fault coverage, the probability of one of several different outcomes as specified by the user, and the time between the occurrence of two events. The results include parameters of the time and probability distribution. Also, confidence intervals are provided for all of the results.

2.5. Conducting a Fault-Injection Campaign with Loki

A fault-injection campaign using Loki comprises three steps. First, the user must provide all necessary input. This includes a description of all of the state machines, probes by which the state machines can interact with the system, descriptions of events that trigger actions, the fault conditionals, and the measures desired. Second, the experiment described must be run. The Loki runtime interacts with the system under study based on the user input to obtain observations about the system. Once the experiment runs are complete, the post-analysis is done. The post-analysis includes synchronization of the timestamps, checking for proper fault injection, and calculating values for the desired measures.

3. CLOCK SYNCHRONIZATION

3.1. Overview

To obtain accurate results about faults, measures, and information regarding the global state of the system, it is necessary that all times recorded regarding these events be based on a single global timeline and that the times be as accurate as possible. To determine information regarding the ordering of events for determination of global state, causal ordering is needed. For measurements such as values for times to cover faults, actual timestamp recordings across the network of nodes are also needed. This chapter discusses how Loki provides a single global timeline. A background of different clock synchronization methods is provided in Section 3.2. Section 3.3 describes the method chosen to provide a global timeline in Loki. This approach is a software algorithm based on message passing. The best way to obtain the necessary data for synchronization of timestamps is discussed in Section 3.4.

3.2. Review of Existing Clock Synchronization Algorithms

Much has been written on how to synchronize clocks to obtain an accurate global time in distributed systems. The primary concerns in choosing a method to synchronize and read global clocks are the accuracy of the reading and the intrusiveness with regard to the system under study. The approaches can be divided into three areas: hardware, software, and some combination of both hardware and software. There are several important tradeoffs to consider when choosing between hardware and software to synchronize clocks and to timestamp important observations and events. One must determine exactly how much intrusion and inaccuracy is acceptable, as well as how portable a solution must be, in order to make a good decision with respect to these tradeoffs.

Hardware clocks are useful for highly accurate clocks and timestamps. Also, by using a hardware interface, the intrusiveness to the system under study is reduced. The disadvantage of using hardware is that such solutions are generally system-specific or will work only with homogeneous systems [11], [12]. One example that provides a more system-independent interface is the *Spearmints* system [13]. Furthermore, most hardware solutions are designed

to work in LAN-type environments. A notable exception is the *CesiumSpray* system that uses GPS receivers to synchronize machines, even across WANs [14].

Systems also exist that use both hardware and software to synchronize globally timed measurements [15]. Using some hardware can help to reduce the intrusion and improve accuracy of some software algorithms. Many software algorithms are limited by the variance in message transmission times. Using specialized hardware can help in limiting this variance.

Many pure software algorithms exist for the synchronization of clocks. If only a global ordering of events or a subset of events that occurred across a distributed platform is needed, logical clocks can be used [16], [17]. If timing of events is necessary, synchronization of all clocks to a global timeline is necessary. One algorithm, by Lamport [16], synchronizes clocks during program execution through the use of messages passed by the system. By observing the messages passed and associated times, the local clocks can be adjusted to a global time. However, if a sufficient number of messages is not passed as the system is executing, extra messages must be passed by the algorithm to ensure accuracy. These extra messages will have the undesired effect of perturbing the system. Also, the accuracy of the clocks is limited by the variance of message transmission delays on the system.

Other algorithms exist that send clock update messages periodically on the system. One proposed method is that all of the nodes of the system periodically request the time from a central server [18]. Another method has nodes broadcast their times to all other nodes at a specified rate [19]. The local nodes update their clocks based on the average of the times received. In both cases, updating the clocks during program execution is intrusive, and the accuracy of the clocks is limited by the message transmission delay time.

3.3. The Loki Clock Synchronization Algorithm

For Loki, the best balance possible of the tradeoffs between intrusion, accuracy, and portability was desired. Loki does not use special hardware to obtain times and measurements. By not using hardware, portability concerns are reduced. Also, the elimination of extra hardware requirements increases Loki's ease of use. A hybrid solution of hardware and software was not integrated into Loki for the same reasons a hardware solution was not used. The increased accuracy of hardware is not needed, but the additional hardware

would detract from usability and portability. While hardware provides the necessary accuracy and limited intrusion, software algorithms exist that are sufficient for our needs. Since Loki is designed for use with LAN testbeds, a system for a WAN, such as *CesiumSpray*, is not necessary.

Since Loki has to report times between events, logical clocks are not sufficient. An algorithm that provides a global clock with event times is required. Since all of the measured event times are compared only to other events that occurred during the experiment, no reference is required to time in the "outside world." In other words, as long as all of the clocks used to timestamp events during a given experiment are synchronized to each other for the duration of the experiment, no further synchronization between experiments or to "wall-clock time" is required.

Any algorithm that updates clocks during program execution will introduce delays into the system. Since Loki only uses the timestamps in calculation of the resulting measures, real-time updating of the clocks is unnecessary. Statistical algorithms that adjust local clock readings after the experiment has completed allow the creation of a global timeline from local clock times without perturbing the system. If implemented correctly, they allow for an arbitrary amount of precision that is independent of the message transport time [20].

The software algorithm that is incorporated in Loki is a post-analysis statistical algorithm [20]. The algorithm adjusts clock readings after the experiment is completed, thus eliminating any intrusion due to synchronization. The precision of this algorithm is not limited by the message delay times. This algorithm is based on the passing of timestamps between all machines for which a single timeline is desired. These messages can be passed at any time when the Loki experiments are not running. Given these timestamps for sending and receiving messages from each machine to every other machine, a linear correction can be calculated for each clock to some reference clock.

3.4. Obtaining Data Points for Synchronization

Having chosen an algorithm to implement a global timeline based on message passing, several decisions must be made. Section 3.4.1 explains and justifies the assumption of a linear clock drift with respect to a reference timeline. Additional information may be used to refine the clock offset to a reference machine. These methods are discussed in Section 3.4.2. Based

on the assumption of linear clock offsets, Section 3.4.3 discusses when to pass messages between machines to get data for an accurate timestamp adjustment as efficiently as possible. The accuracy of the global timeline depends on the accuracy of local clock readings. Section 3.4.4 describes the method used to get precise readings from the local clocks. The precision of the mathematical operations used to calculate the clock adjustments also has to be sufficient to make a precise global timeline possible. This is discussed in Section 3.4.5.

3.4.1. Representing time offsets

When adjusting a local timestamp to a global timeline *t*, consider *C(t)* as a function that maps the local clock to be synchronized to the global continuous time. If *C(t) = t*, the clock perfectly mirrors the global time. This is called a *perfect* clock and is equivalent to the reference clock. The difference between any clock and a perfect one can be represented as [21]

$$t - C(t) = \Delta t = \alpha(t_0) + \beta(t_0)(t - t_0) + \delta(t - t_0)^2 + \varepsilon(t)$$

where Δt is the offset at time *t*, $\alpha(t_0)$ is the initial offset error, $\beta(t_0)$ is the frequency offset, δ is the frequency offset rate, and $\varepsilon(t)$ represents other errors.

According to [21], δ and ε are small enough for crystal devices such as those used in computer systems that they may be considered zero. This is because the effects of crystal aging and temperature dependence are the most significant components of δ and ε. Over the duration of the experiment, the temperature should not change significantly. Furthermore, the length of the experiments is not long enough for any significant effects from crystal aging. Thus, for our purposes, the difference equation can be rewritten as

$$\Delta t = \alpha + \beta * t$$

The difference between two machine clocks is a linear difference, represented by a fixed offset α and an offset rate β. For each α and β calculated, there is some level of uncertainty. To quantify this uncertainty, limits must be associated with both α and β. Minimum and maximum values (α^-, α^+, β^-, and β^+) are calculated for the offset and offset rate associated with each adjusted clock. These limits can be either confidence intervals or physical bounds.

3.4.2. Constraints to refine offsets

These results are based on adjusting each of the local clocks to one reference clock. While this provides adequate bounds for distinguishing all events in our test cases, tighter bounds may be desired for more precise results from the measure estimation portion of Loki. One way to obtain tighter bounds is to adjust the number of data points and length of message-passing periods used in calculation of the bounds. This is described in Section 3.4.3. Another way is to use additional information about the timestamps and data collected, which is described below.

Using messages passed between two nonreference machines

Since all of the clocks are referenced to one reference clock, each clock is scaled based only on messages passed between it and the reference machine. If messages are passed between each combination of machines, more information is available. For instance, assuming a three-machine network with reference machine *i* and two other machines *j* and *k*,

$$\alpha_{ij}^{-} \le \alpha_{ij} \le \alpha_{ij}^{+} \qquad \beta_{ij}^{-} \le \beta_{ij} \le \beta_{ij}^{+}$$

$$\alpha_{ik}^{-} \le \alpha_{ik} \le \alpha_{ik}^{+} \qquad \beta_{ik}^{-} \le \beta_{ik} \le \beta_{ik}^{+}$$

$$\alpha_{jk}^{-} \le \alpha_{jk} \le \alpha_{jk}^{+} \qquad \beta_{jk}^{-} \le \beta_{jk} \le \beta_{jk}^{+}$$

where α_{ij}^{-} represents the minimum α limit from machine *j* to machine *i*, β_{jk}^{+} represents the maximum β limit from machine *k* to machine *j*, and so forth. After some algebraic manipulations we obtained the following equations:

$$\beta_{jk} = \beta_{ji}\beta_{ik} \qquad \beta_{ji} = \frac{1}{\beta_{ij}}$$

$$\alpha_{jk} = \alpha_{ik} + \beta_{jk}\alpha_{ji} \qquad \alpha_{ji} = -\alpha_{ij}$$

Based on these equations and inequalities, limits can be further reduced between the reference machine *i* and machines *j* and *k*. For example, if the limit $\beta_{ik}^{+} \ge \beta_{jk}^{+} * \beta_{ij}^{-}$, then the limit can be reduced to $\beta_{jk}^{+} * \beta_{ij}^{-}$. The formal equations to reduce the limits based on three machines are as follows:

$$\beta_{ik}^{-} = \max(\beta_{ik}^{-}, \beta_{jk}^{-}\beta_{ij}^{+}) \qquad \beta_{ik}^{+} = \min(\beta_{ik}^{+}, \beta_{jk}^{+}\beta_{ij}^{-})$$

$$\beta_{ij}^{-} = \max(\beta_{ij}^{-}, \frac{\beta_{ik}^{+}}{\beta_{jk}^{+}}) \qquad \beta_{ij}^{+} = \min(\beta_{ij}^{+}, \frac{\beta_{ik}^{-}}{\beta_{jk}^{-}})$$

$$\alpha_{ik}^{-} = \max(\alpha_{ik}^{-}, \alpha_{jk}^{-} + \frac{\beta_{ik}^{+}\alpha_{ij}^{+}}{\beta_{ij}^{-}}) \qquad \alpha_{ik}^{+} = \min(\alpha_{ik}^{+}, \alpha_{jk}^{+} + \frac{\beta_{ik}^{-}\alpha_{ij}^{-}}{\beta_{ij}^{+}})$$

$$\alpha_{ij}^{-} = \max(\alpha_{ij}^{-}, (\alpha_{ik}^{+} - \alpha_{jk}^{+})\frac{\beta_{ij}^{+}}{\beta_{ik}^{-}}) \qquad \alpha_{ij}^{+} = \min(\alpha_{ij}^{+}, (\alpha_{ik}^{-} - \alpha_{jk}^{-})\frac{\beta_{ij}^{-}}{\beta_{ik}^{+}})$$

$$\beta_{ik}^{-} = \max(\beta_{ik}^{-}, (\alpha_{ik}^{+} - \alpha_{jk}^{+})\frac{\beta_{ij}^{+}}{\alpha_{ij}^{-}}) \qquad \beta_{ik}^{+} = \min(\beta_{ik}^{+}, (\alpha_{ik}^{-} - \alpha_{jk}^{-})\frac{\beta_{ij}^{-}}{\alpha_{ij}^{+}})$$

$$\beta_{ij}^{-} = \max(\beta_{ij}^{-}, \frac{\beta_{ik}^{+}\alpha_{ij}^{+}}{\alpha_{ik}^{-} - \alpha_{jk}^{-}}) \qquad \beta_{ij}^{+} = \min(\beta_{ij}^{+}, \frac{\beta_{ik}^{-}\alpha_{ij}^{-}}{\alpha_{ik}^{+} - \alpha_{jk}^{+}})$$

To fully reduce the limits for three or more machines, one can simply compare inequalities between any two machines (doing so for all possible pairs of machines) and repeat the comparisons until the limits do not change.

Causal information

Another possibility for reducing the bounds on timestamps is the use of causal information. If the bounds for the times of two events overlap, the bounds may be reduced by knowledge of the causal order of the events. For example, if two events occur on different machines with overlapping bounds, yet one event caused the other, the minimum bound of the second event must be no less than the minimum bound of the preceding event. Conversely, the maximum bound of the first event cannot be greater than the maximum bound of the second event.

3.4.3. When to obtain data points

Since messages are passed and recorded in a normal fault-injection experiment, the messages themselves could be used to calculate the adjustments. However, it cannot be guaranteed that enough messages will be passed to obtain the desired precision for the global timeline. For example, one machine might get update messages from other machines but never send any of its own. Without messages sent from a machine, it is impossible to calculate the adjustment for its timeline. Also, since additional information about the local

timestamps would need to be added in normal messages, the program execution would be perturbed. Since there may not be enough messages passed between all machines, and perturbation is not desired, all messages used for timestamp synchronization are sent and received during times when the fault-injection experiment is not under way. Messages could be sent before or after the experiments. Due to the linear nature of the clock error, the error in the calculated estimation would grow as adjusted times grew farther away from the time when the synchronization messages were passed. This is shown in Figure 2. This method will work for short experiments, but as experiments grow longer, the length of the message-passing phase necessary to reduce the error will become prohibitively long.

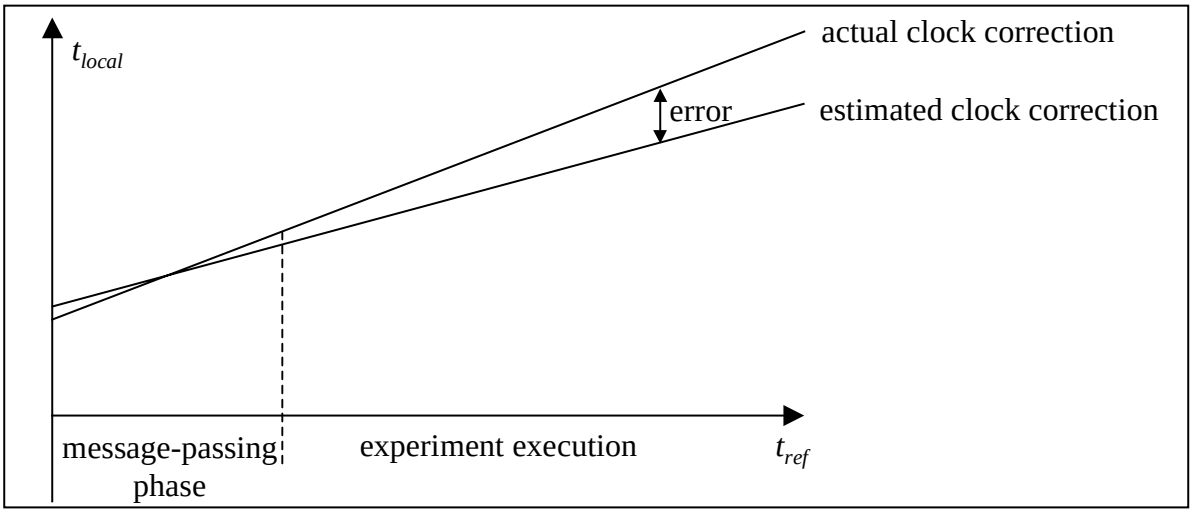

Figure 2: Estimation Error Grows as Experiment Time Grows

The solution to this problem is to obtain data points for synchronization both before and after the experiment [22]. This will eliminate the possibility of performing on-line adjustment of timestamps. However, that is not a concern in our case anyway since the global timeline is only used in postanalysis. Because sampling is done both before and after the experiments, the error in clock adjustment cannot grow without bound during the experiments. This can be seen in Figure 3.

The “before and after” method allows for global timeline estimation over an arbitrary length of time. Better yet, the longer the experiment takes, the greater the accuracy of the estimation algorithm, as will be seen in Chapter 4. There are three factors that influence the

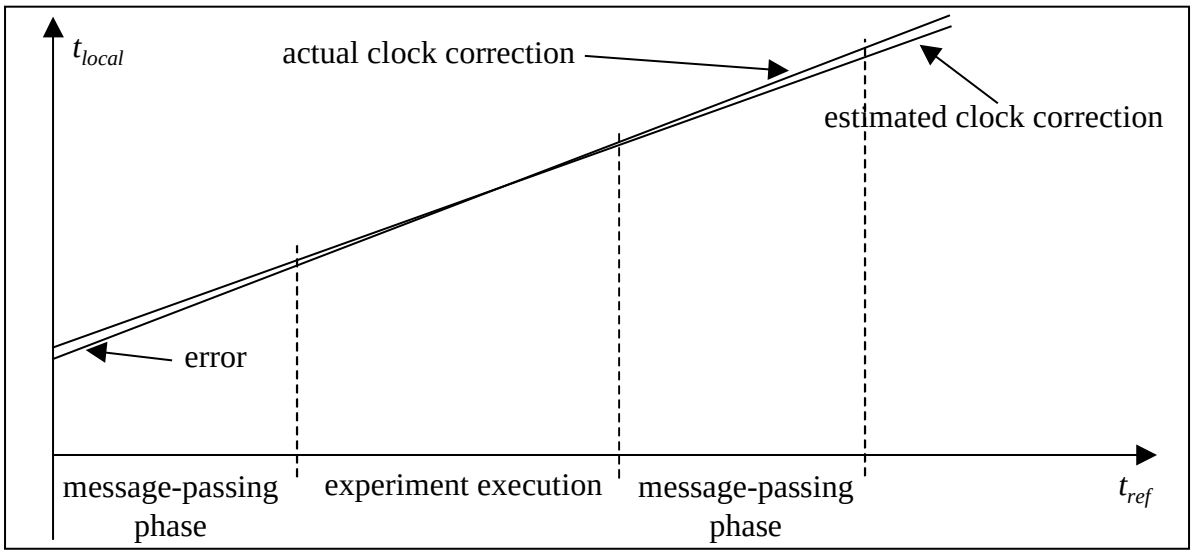

Figure 3: Passing Messages Before and After Experiments Limits Estimation Error

accuracy of the offset estimation. First, increasing the length of the message-passing phase increases the accuracy of the results. Second, the more messages that are passed during each message-passing phase, the more accurate the results will be. Finally, the longer the experiment takes, the better the estimation will be. For more details on how accurate the global timeline estimations are based on how these three factors are varied, see Chapter 4.

3.4.4. Obtaining accurate readings

A time measurement can only be as accurate as the call used to read the time measurement. The POSIX system call `gettimeofday()` provides relatively portable access to a system clock on UNIX-like systems. However, the accuracy of the clock is limited by the operating system, since system calls must wait for processing by the operating system before the clock is read. Thus, the accuracy cannot be guaranteed to be better than that of the "heartbeat" of the operating system. The *heartbeat* is the predefined period after which the processor must switch out any thread currently executing and process any interrupts. The standard heartbeat on Linux, the operating system on which Loki was developed, is 10 ms. Although the time to return from the system call is usually much less than 10 ms, a clock that is not dependent on system calls was desired.

On-chip cycle counters can provide readings that are independent of system calls, and thus more accurate. For the Pentium and newer processors from Intel, the `rdtsc` (read

timestamp counter) instruction is an on-chip counter that can be used to obtain accurate clock readings [23]. This instruction reads a 64-bit value containing the number of clock cycles that have taken place since the machine was powered on. The size of the counter guarantees that the counter will not roll over during experiments.

The counter is accessible for reading with user privileges, so Loki needs no special privileges to read the counter. Since `rdtsc` is an assembly instruction executed in user space, it doesn't have the overhead of system calls like `gettimeofday()`, and hence can provide a quicker clock reading. Thus, the accuracy of the recording is limited by the same factors as any outside hardware clock, namely, the time it takes the processor to switch in the instructions from the hardware clock or from the `rdtsc` instruction. The disadvantage of this instruction is that it is limited to specific processors, and thus is not totally portable. For systems without access to the `rdtsc` instruction, the less accurate `gettimeofday()` system call is used.

The processors known to support the `rdtsc` instruction include Pentium or newer processors from Intel, K5 and newer processors from AMD, and the 6x86MX and GXm processors from Cyrix. There are other processors that also include on-chip counter instructions, so if instructions to access these counters are included in Loki portability will be increased. Such instructions include the `rpcc` (read performance cycle counter) instruction on the DEC Alpha.

3.4.5. Precision

When implementing the clock adjustment algorithms, it was found that the standard 64-bit double precision arithmetic was not sufficient to produce results as accurate as desired. An initial implementation of the algorithm using standard data types produced incorrect results. This is because the data points being used could be very large (with time measured either in μs or in processor clock ticks), while the differences among these data points could be very small. It must be noted that data point values are large even though the local clock readings are all adjusted so that the experiment starts at time zero. To overcome this limitation, a new data type was created on which more precise mathematical operations could be conducted.

The new data type consists of two parts, the whole part and the fractional part. Each part consists of 2 `unsigned long` values to hold the number and a `char` to hold the appropriate sign. This totals 80 bits to hold a value instead of 64 bits. While the additional space necessary is not too great, the computation time increases significantly for each mathematical operation. This is because the operations must be performed in software instead of hardware. The mathematical functions implemented for this data type include several inequality comparators, addition, subtraction, multiplication, and division.

The comparator functions are straightforward. These functions simply compare the signs and then, as necessary, compare the absolute values from greatest magnitude to least. Similarly, with addition and subtraction, signs are compared, and then addition or subtraction is performed on each of the values, from least magnitude to greatest, carrying the bits as necessary between the `long` values. Multiplication and division are slightly more complex. For all arithmetic needed by the new data type, divisors are always whole numbers; so is the multiplier for any multiplication. These restrictions made the mathematics much easier. Both of these functions are based on algorithms in [24]. The algorithms are in Appendices A and B.

4. TIMELINE ESTIMATION

4.1. Overview

Chapter 3 discussed the determination of the best methods for obtaining clock readings and passing messages for Loki. The data can now be used to synchronize the local clock readings to a single timeline. To calculate α and β, the offset and slope of clock drift, several different approaches were tried. All three of the approaches discussed adjust all of the local clock readings to a single reference timeline by comparing times between the local clock and the reference clock. The first two methods produce confidence intervals for α and β. The first approach studied uses regression analysis (Section 4.2). Because that approach may result in negative message delays, a modified version of the method using constraints was implemented instead (Section 4.3). Section 4.4 discusses the use of a convex hull algorithm that produces physical bounds for the adjustments calculated.

4.2. Regression Analysis

As mentioned previously, clocks are synchronized by the passing of messages between machines and the recording of the send and receive times. This is done both before and after the experiments have taken place. Figure 4 shows a plot of send and receive times between a reference machine and a machine for which the clock is to be adjusted.

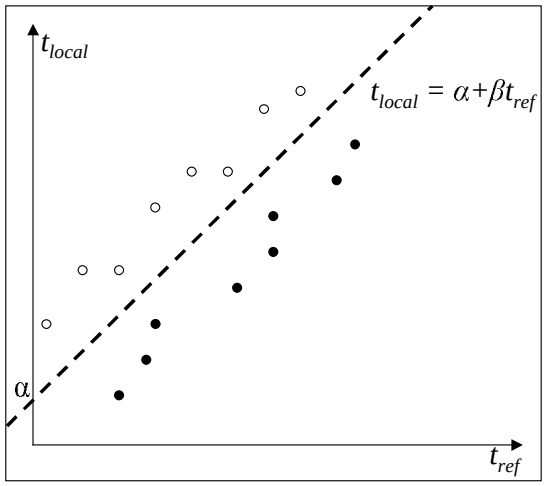

Figure 4: Plot of Timestamps and Approximate Linear Adjustment

The line in Figure 4 represents a best fit of the plotted timestamps based on a minimum sum of squares optimization used for regression analysis. The points above the line (the light circles) represent messages sent from the reference machine (on the t_{ref}-axis) to the local machine (on the t_{local}-axis). Points below the line (the dark circles) represent messages sent from the local machine (on the t_{local}-axis) to the reference machine (on the t_{ref}-axis). The gaps between these data points and the line represent the delay times that the messages spend on the network.

Simple linear regression calculates a best fit line for the dependent variable (t_{local}) based on the independent variable (t_{ref}). It assumes that the error between the best fit and data points will have a normal distribution. The line calculated can thus be described as

$$t_{local} = \alpha + \beta * t_{ref}$$

where α and β are calculated based on the following equations:

$$\hat{\beta} = \frac{\sum_{i=1}^{n} t_{ref(i)} t_{local(i)} - \frac{\sum_{i=1}^{n} t_{ref(i)} \sum_{i=1}^{n} t_{local(i)}}{n}}{\sum_{i=1}^{n} t_{ref(i)}^2 - \frac{(\sum_{i=1}^{n} t_{ref(i)})^2}{n}} \qquad \hat{\alpha} = \overline{t_{local}} - \hat{\beta} * \overline{t_{ref}}$$

Given the assumptions used in applying simple linear regression (stated above), the estimators of α and β will also be normally distributed [25].

To obtain values for α and β, a regression analysis routine, *nag_simple_linear_regression*, from the Numerical Algorithms Group (NAG) was used [26]. This routine calculated the values for α and β, but did not put any constraints on the resulting line. Without any constraints, regression analysis has a major drawback. If any of the dark circles lie above the line, or any of the light circles lie below the line, the implication is that, on the adjusted clock, that message was received before it was sent. Since that is not acceptable, the best fit algorithm must constrain the line such that all of the light circles lie above the line and all of the dark circles lie below the line. This is accomplished using a constrained optimization routine.

4.3. Constrained Optimization Problem

The algorithm used to calculate α and β with the constraints on the delays was a minimization with nonlinear constraints using a sequential quadratic programming method. This was done using routine *nag_opt_nlp* from NAG. The function to minimize is a sum of least squares. By minimizing the sum of the distance of all of the data points from the line, a best curve fit is found. The constraints are that all of the data points associated with messages sent from the reference machine to the local machine must lie above the line, while the messages sent from the other machine to the reference machine must lie below the line.

Analytical expressions of confidence intervals for the timestamps can be found only for some standard distributions. The assumption made in obtaining these confidence intervals is that the delays follow a normal distribution. If the resulting offsets have a normal distribution, and clocks drift linearly, it is likely that $\hat{\alpha}$ and $\hat{\beta}$ are also normally distributed. Using these assumptions, confidence intervals for timestamps can be calculated.

After observing message delay times and running the Shapiro-Wilk and Kolmogorov-Smirnov tests for normality, it was determined that the message passing time distribution was far from being a normal distribution. This was true with the use of both `gettimeofday()` and `rdstc` calls to read clocks, although the resulting distributions were significantly different.

As can be seen in Figure 5, the distribution of delay times using the `gettimeofday()` system call does not fit any simple distribution. The multimodal nature of the distribution is most likely due to the overhead of using a system call. It is this overhead that causes the mean recorded transmission time to be much greater than that for `rdstc` (Figure 6) and explains the multimodal distribution.

The results shown in Figure 5 motivated the use of a clock observation mechanism that was not dependent on system calls. Using the `rdstc` instruction improved the accuracy of the clock readings by reducing mean recorded transmission time, and also changed the distribution of recorded message passing transmission times. The resulting distribution is shown in Figure 6. Although the distribution is different, it is still a nonstandard multimodal distribution.

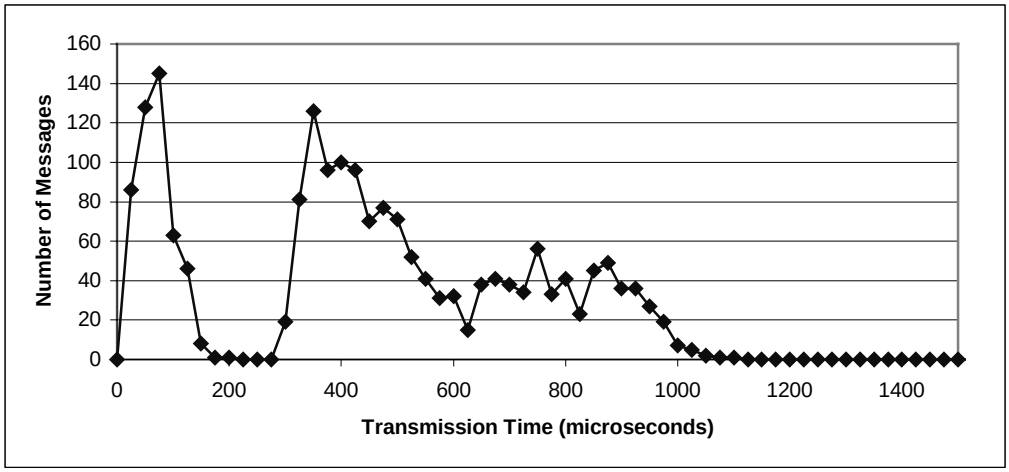

Figure 5: Distribution of Delay Times with `gettimeofday()` System Call

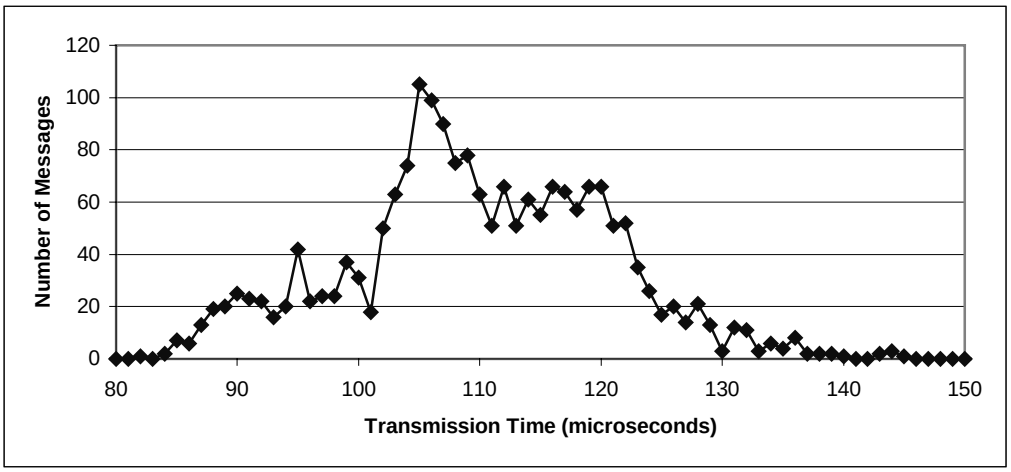

Figure 6: Distribution of Delay Times with `rdtsc` Instruction

Although confidence intervals could not be calculated due to the nonnormal distributions, tests were run to determine clock offsets. In every case tested, the results from the constrained optimization routine were the same as those from the simple linear regression analysis routine.

4.4. Convex Hull

Since the recorded message delays are of a nonstandard multimodal distribution, calculation of confidence intervals is not practically possible. The other option is the use of absolute bounds for each timestamp based on the condition of positive message transmission times, employing the use of convex hulls.

The dark lines of the convex hulls in Figure 7 represent the physical bounds for each timestamp. For messages passed between two machines, the upper hull represents messages sent from the reference machine to the local machine. The upper hull defines the lower bound of an adjusted timestamp on the global timeline. The bottom hull represents messages sent from the local machine to the reference machine. The lower hull defines the upper bound of an adjusted timestamp on the global timeline. Determination of the upper and lower hulls is described in Section 4.4.1.

The dashed lines in Figure 7 represent the absolute limits for any α and β based on the constraint of linear clock offsets. One line represents the maximum β (β_{max}) and minimum α (α_{min}), while the other line represents the minimum β (β_{min}) and maximum α (α_{max}). The determination of the limits on α and β is described in Section 4.4.2. It is important to note that the linear constraints α_{min}, α_{max}, β_{min}, and β_{max} are referred to as *limits* in this document. These limits, as well as the convex hulls, are utilized to produce *bounds* for all timestamps. The timestamp bounds based on values from the convex hull may be reduced using the limits imposed by the linear drift assumption. In particular, this is done using the fact that the α and β for lines defining the convex hull cannot exceed the minimum and maximum limits determined by the linear offset constraint. Determination of bounds for a given timestamp based on the convex hull and limits on α and β is described in Section 4.4.3.

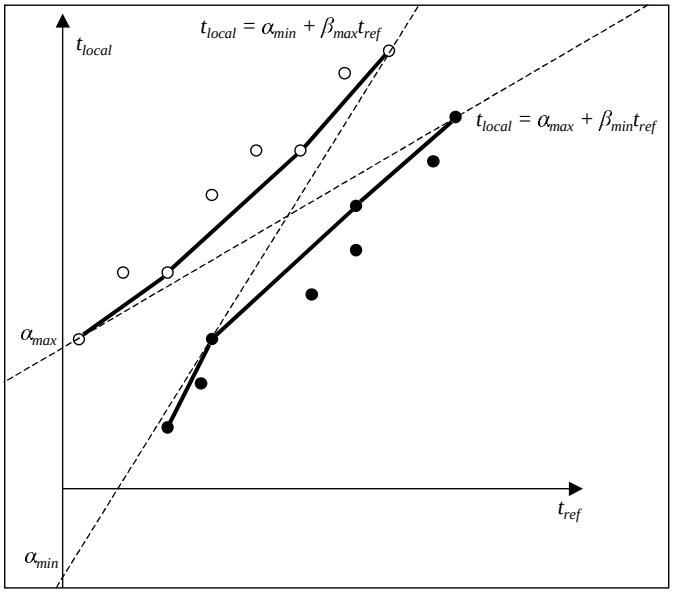

Figure 7: Convex Hull with Limiting Linear Adjustments

The determination of the two convex hulls is based on a "package-wrapping" algorithm [27]. This algorithm is illustrated in Figure 8. The first step of the package-wrapping algorithm is to find a starting point on the hull. This can be done by simply finding a minimum or maximum *x*- or *y*-coordinate. Any point satisfying this condition must be on the hull. Then, additional points on the hull can be successively determined by taking a line anchored to the last point found and sweeping it around until a point is encountered. This point must fall on the hull. This is done until the first point determined to be on the hull is encountered again.

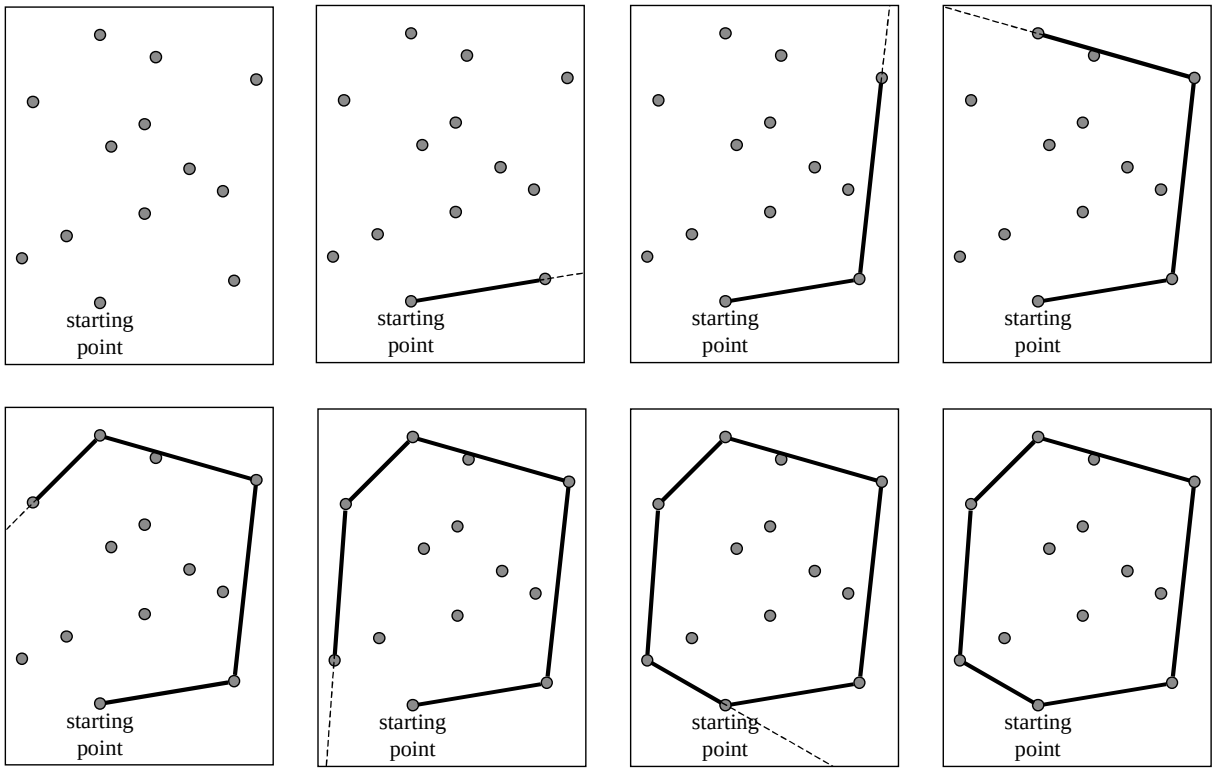

Figure 8: Package-Wrapping Algorithm

4.4.1. Determination of upper and lower hulls

Several modifications have been made to the package-wrapping algorithm so that it can be more easily applied to the message passing times. In particular, since only the section of each hull that defines the region of valid global time is of interest, only that half of the top and bottom hull need be defined.

Figure 9 illustrates the process for determining the upper hull. Definition of the top hull starts by choosing the first message sent and received among light points used in determining the upper hull (since it must have both the minimum *x*- and *y*-coordinates) as the first point on the hull. Then, points on the hull can be determined by sweeping a line in a counter-clockwise direction. This is continued until a point is found that has a *y*-coordinate that is less than the previous point (past the top of the hull). This point is on the side of the hull that does not define the region of valid time. Therefore, the partial-hull needed is fully defined by all of the previous points.

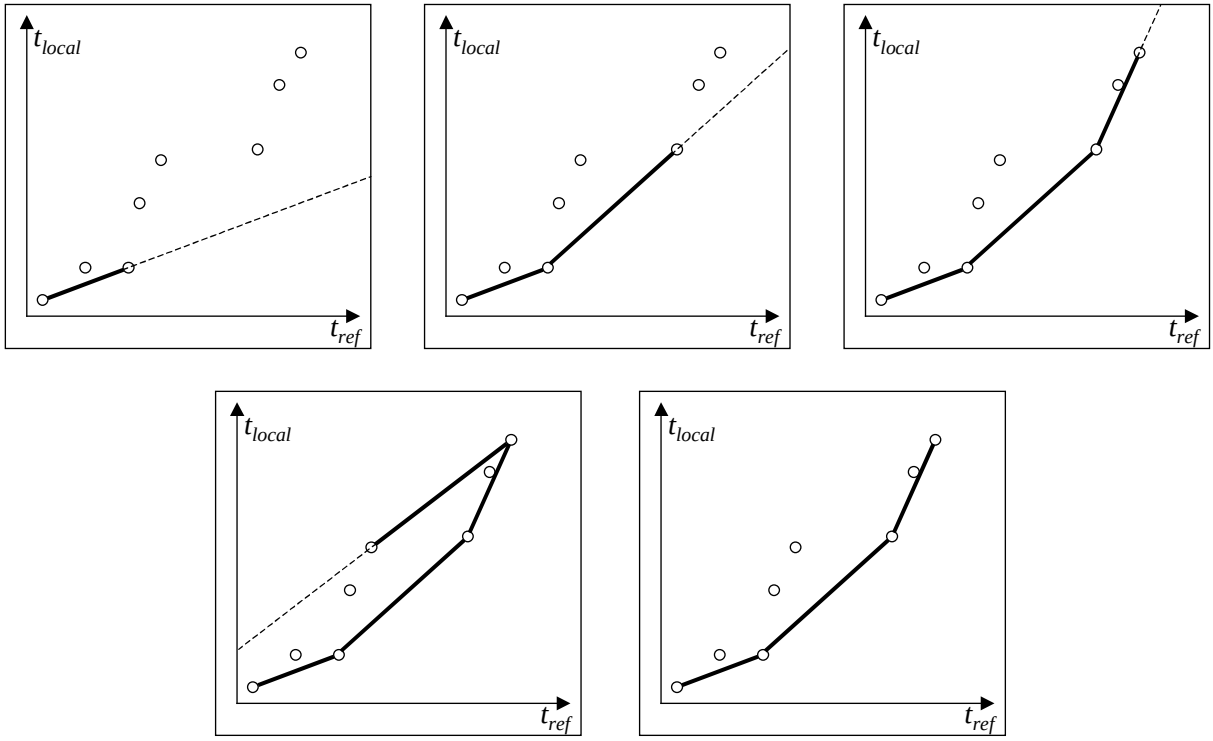

Figure 9: Determination of Upper Hull

For the bottom hull, the first message sent and received among dark points used in defining the bottom hull can be chosen as the first point on the hull. However, if a counter-clockwise sweep is used to find points in the bottom hull, the wrong region of the hull will be defined. Instead, the sweep must proceed in a clockwise direction to define the hull region that determines the region of valid times. As with the top hull, this is done until a point is reached that has a *y*-coordinate less than that of the previous point. This process is illustrated in Figure 10.

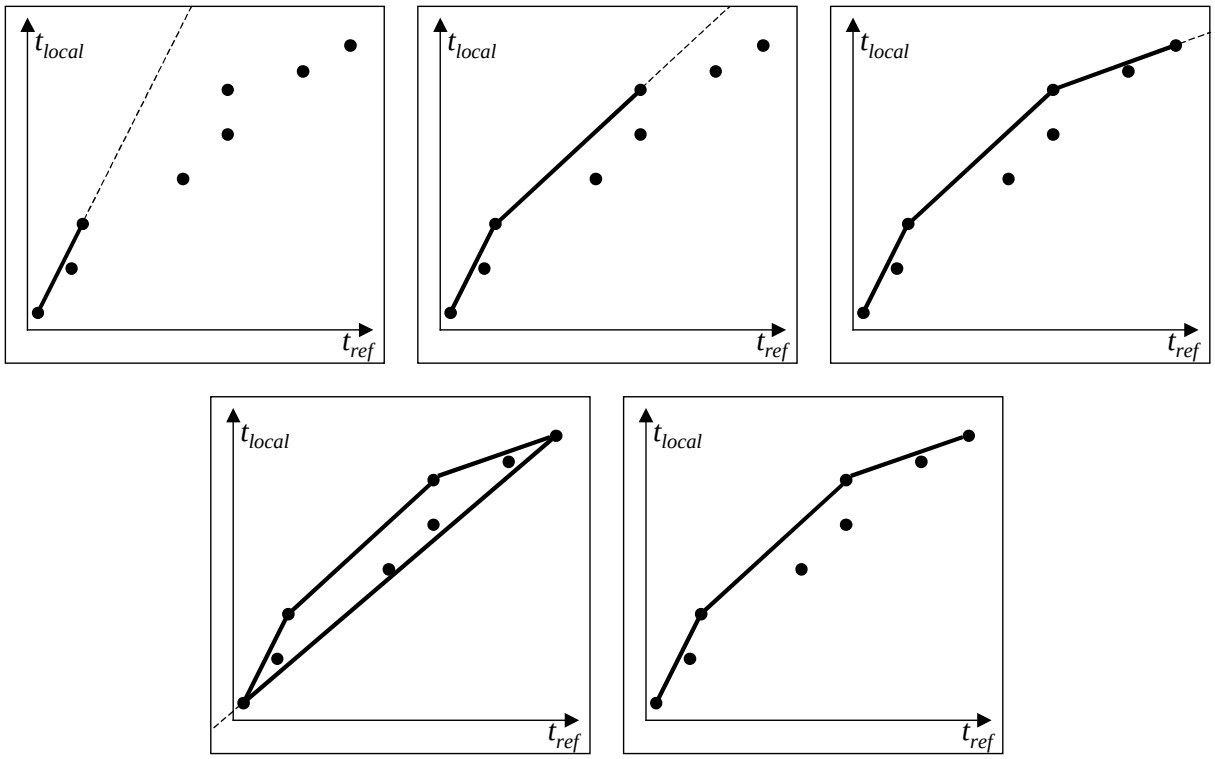

Figure 10: Determination of Lower Hull

4.4.2. Determination of limits for α and β

After the convex hulls that define the region of valid times have been determined, the lines that represent the limits on α and β for a clock with respect to the reference machine (assuming liear drift) must next be determined. These are the dotted lines shown in Figure 7. Those lines are calculated to determine the physical maximum and minimum α and β. This section describes how α_{min}, α_{max}, β_{min}, and β_{max} are calculated.

To determine the line that constitutes α_{min} and β_{max}, one starts by finding the line defined by the first two points on the lower convex hull. If any point on the upper hull occurs below this line, one moves to the line defined by the second and third points on the lower hull. If any points on the upper hull fall below this line, one moves to the next set of points on the lower hull. This is done until a line defined by two points on the lower hull lies completely below the upper hull. This is shown in Figure 11. The lower of these two points on the lower hull defines one point on the line defining α_{min} and β_{max}. This point is determined to be *pivot_1* in the right half of Figure 11. It is theoretically possible for no line to exist that will not cross the upper hull. However, this is not of practical concern, because the hulls will be

nearly straight lines, and the transmission time gap between the hulls is large enough to assure that there will always be a line that does not cross into the upper hull.

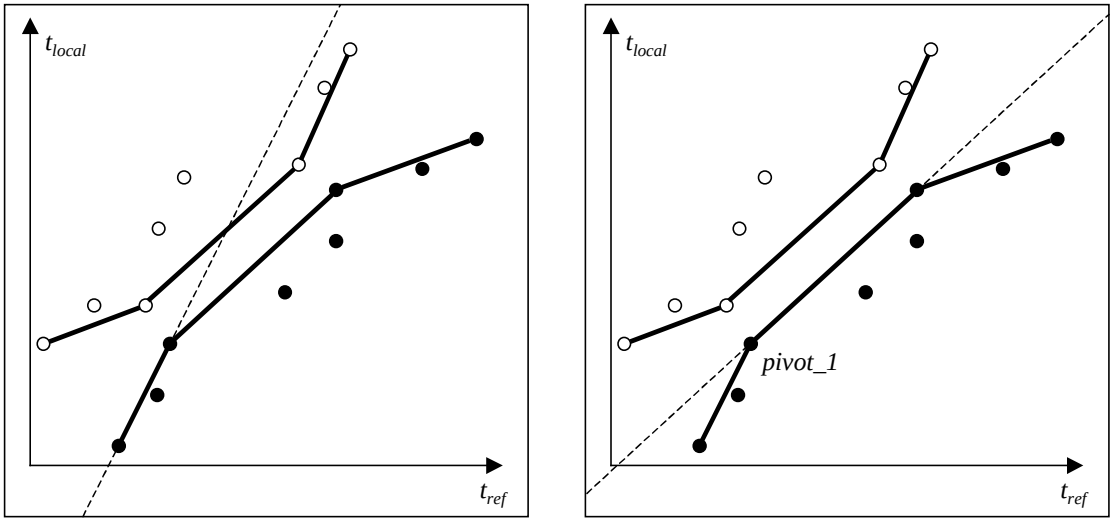

Figure 11: First Half of Finding α_{min} and β_{max}

Next, the algorithm checks for any point on the upper hull that lies below a line defined by *pivot_1* and the highest point on the upper hull. This is shown in Figure 12. If any point on the hull is below the line, the algorithm checks whether any point on the upper hull lies below a line defined by *pivot_1* and the next highest point on the upper hull. Checking is continued until no point on the upper hull lies below the line in question. This line, determined by *pivot_1* and *pivot_2*, is tangential to both the upper and lower hulls. This line, shown in the rightmost picture of Figure 12, defines the α_{min} and β_{max}.

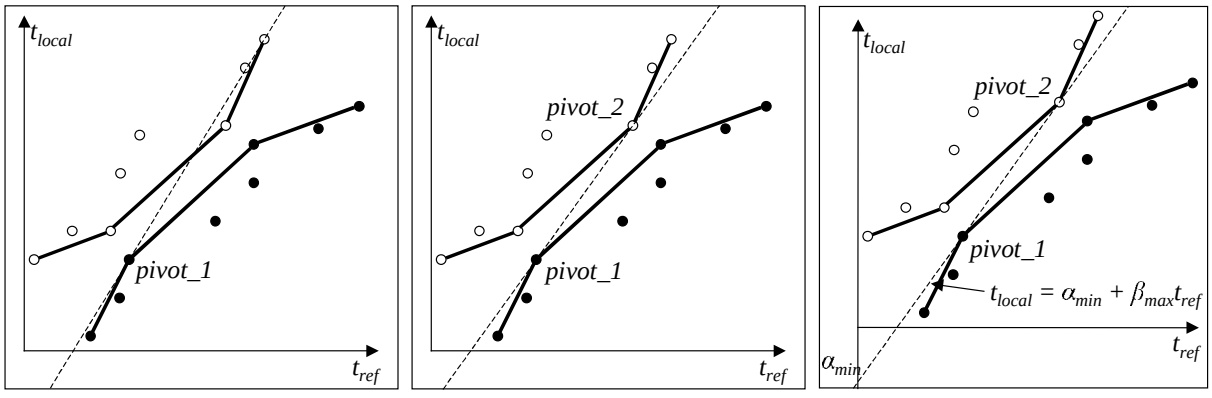

Figure 12: Second Half of Finding α_{min} and β_{max}

A similar method is used to determine the line that constitutes α_{max} and β_{min}. Points are found on the upper hull that determine a line that does not intersect the lower hull (Figure 13).

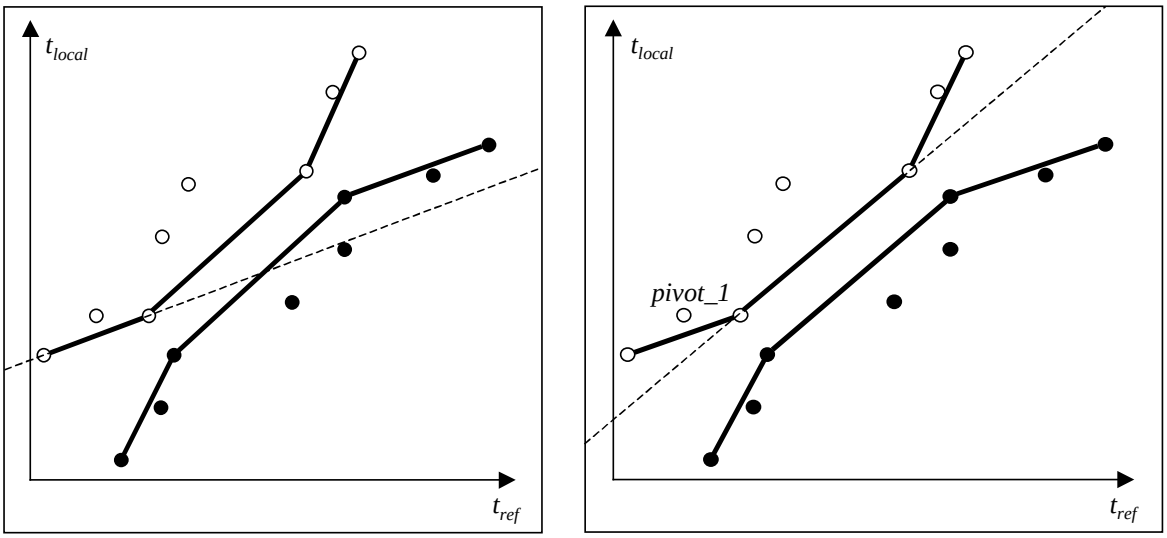

Figure 13: First Half of Finding α_{max} and β_{min}

Once this line is determined, the point *pivot_1* is used as an anchor to determine a line with a point on the lower hull that is tangential to both the lower and upper hulls (Figure 14). This line is shown in the rightmost picture of Figure 14. It is determined by *pivot_1* and *pivot_2* and defines α_{max} and β_{min}.

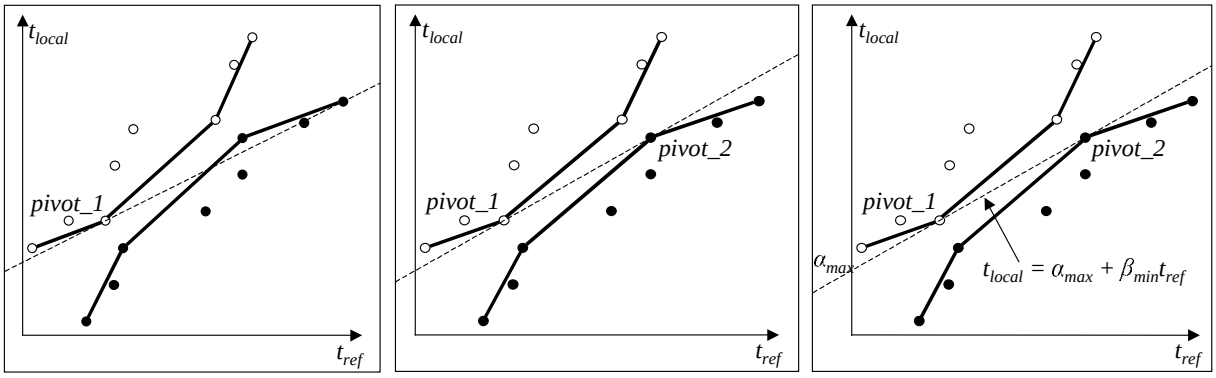

Figure 14: Second Half of Finding α_{max} and β_{min}

4.4.3. Bounding timestamps

As mentioned previously, the limits on α and β are used to refine the bounds on timestamp adjustments. The limits do not give us the bounds on the global timestamp values by themselves. If just the lines were used for determining the bounds, the minimum and maximum timestamp bounds would be equal at the point where the lines intersect, as seen in the left half of Figure 15. This is obviously not right. Nor should the lines defined by α_{min} and β_{min}, and by α_{max} and β_{max}, be used. These lines, shown in the right half of Figure 15, diverge with bounds getting worse and worse as time grows.

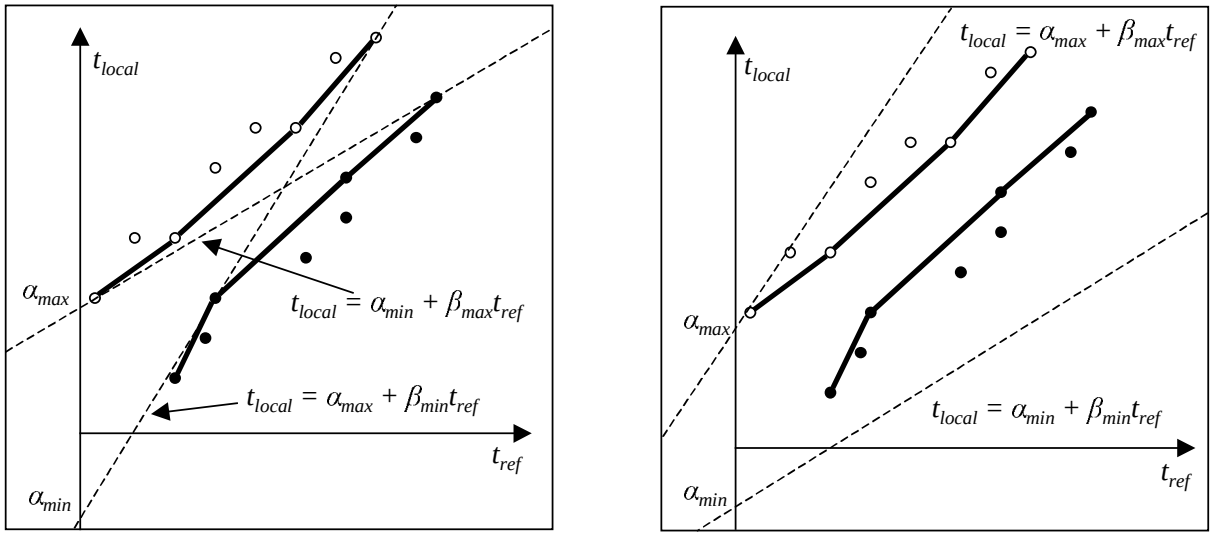

Figure 15: Incorrect Limits for α and β--Intersecting Lines and Diverging Lines

The points on the upper and lower hulls that will correspond to the upper and lower bounds for timestamps must be found. These hulls are the dark lines in Figure 16. To find the corresponding value on a hull, the bound value is calculated for each line defined by adjacent points on the hull. On the upper hull, the minimum of these values defines the correct minimum bound. On the lower hull, the maximum of these values defines the correct maximum bound. Only lines with slopes and offsets that do not violate the α and β limits are considered. For instance, hull segment 1 on the lower hull in Figure 16 cannot be considered because it falls outside the limits imposed by β_{max} and α_{min}.

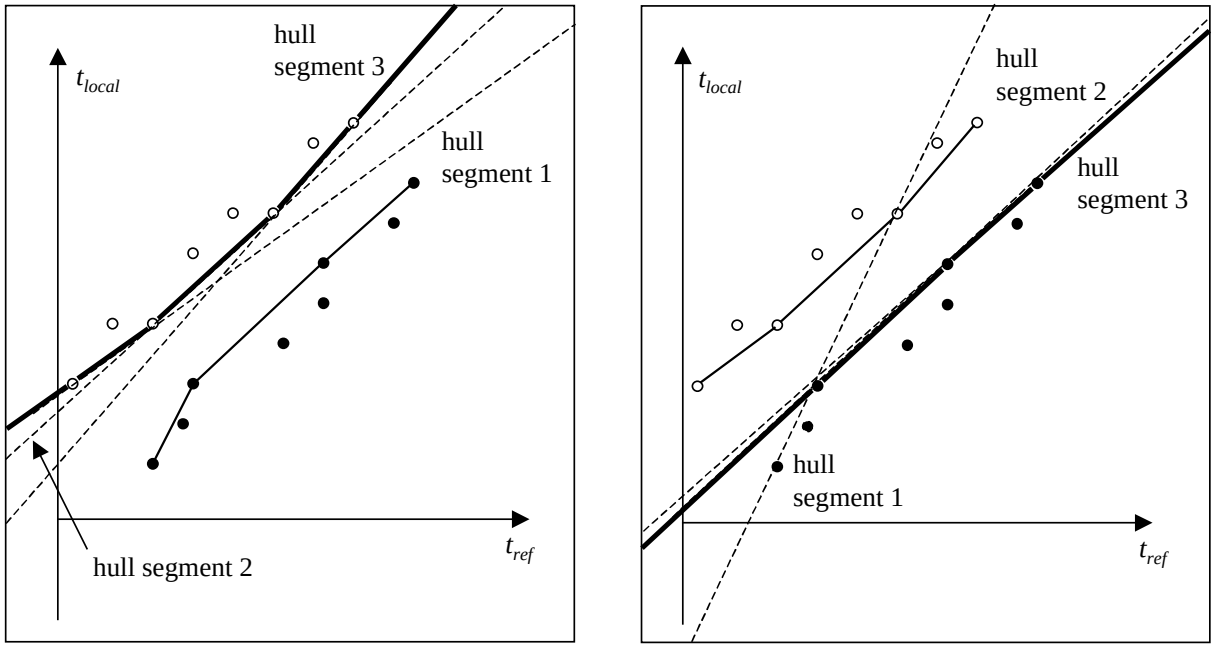

Figure 16: Calculating the Bound on the Upper Hull and on the Lower Hull

These timestamp bounds must be further refined by being checked against the lines defining the absolute limits on α and β. If either of the values from the defining limit lines is greater than the previous maximum value, then it is the new maximum value. Similarly, if either of the values from the defining limit lines is less than the previous minimum value, then it is the new minimum value. Figure 17 illustrates how the bounds are refined. The dark lines illustrate the bounds for timestamps. The line segments that extend from points on the hull to the edge of the picture are all based on the limiting α and β.

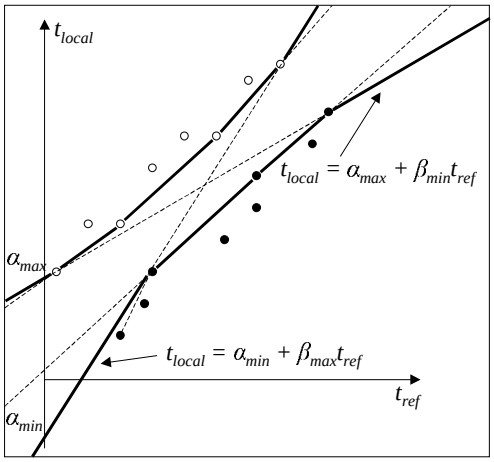

Figure 17: Adjustment of Bounds Based on Limits of α and β

Two example bounds are shown in Figure 18. Time *t1* on the local clock is adjusted to the bounds of $t1_{min}$ and $t1_{max}$. The value of $t1_{min}$ lies on the upper convex hull, while $t1_{max}$ lies on the line defined by α_{max} and β_{min}. The second example time *t2* is adjusted to $t2_{min}$ on the upper hull and $t2_{max}$ on the lower hull.

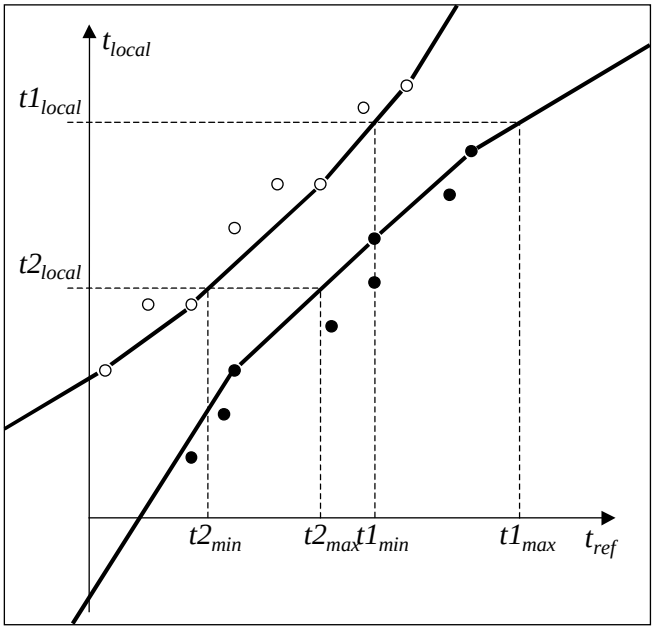

Figure 18: Example Timestamp Adjustment

4.4.4. Accuracy of convex hull method

The accuracy of the bounds calculated using the convex hull method depends on several factors. The accuracy varies depending on the length of the experiment, the length of message passing intervals, and the number of messages passed during each interval. Testing of the accuracy of the limits for α and β was conducted on five Pentium II 233-MHz machines running LinuxOS kernel 2.0.32. The machines were connected by a 100-MB/s Ethernet. Messages were passed between five machines, which resulted in the determination of α and β limits for 10 pairs of machines. In each case, one of the machines was deemed the reference machine, while the other was the machine whose clock readings were to be adjusted. The determination of limiting α and β was conducted 10 times for each configuration tested, resulting in 100 trials for each mean value reported.

Table 1 shows how the range of the limits on α and β changes as the delay between collection of the first (before the experiment) and second (after the experiment) groups of data

points changes. This delay between collecting data points is equivalent to the time it takes to conduct the experiment. The length of the experiment was varied from 7.5 min to 1 hr; the message-passing interval was fixed at 2.5 min for both before and after the experiment; and the number of messages passed was fixed at one per second. The results in Table 1 show that as the experiment gets larger and takes more time, the limits get tighter. The difference in limiting β versus the length between batches of gathering data points is approximately inversely linear in proportion. The difference in limiting α versus experiment time improves much more slowly than does the β difference.

Table 1: Difference in Limits versus Delay Time

Length of Experiment (min)	Difference in α_{min} and α_{max} (μs)	Difference in β_{min} and β_{max} ($x10^{-7}$)
7.5	234.81	6.247
15	233.94	3.869
30	225.75	2.142
60	223.96	1.140

Another variable that can affect the tightness of the limits is the time interval for passing messages both before and after the experiment. To test the influence of message passing interval length, the interval was varied from 1 to 10 min both before and after the experiment; the length of the experiment was held at 7.5 min; and the number of messages passed was fixed at one per second. As can be seen in Table 2, the longer that messages are passed, the better the results. However, the effect on reduction of the difference in limits versus the increase in time to pass messages is not nearly as significant as the effect of increasing time between message passing batches.

Table 2: Difference in Limits versus Time to Pass Messages

Message Passing Interval (min)	Difference in α_{min} and α_{max} (μs)	Difference in β_{min} and β_{max} ($x10^{-7}$)
1	222.64	7.733
2	222.45	6.371
5	221.20	4.162
10	219.40	2.638

Increasing the number of messages passed, and thus the number of data points used in the convex hull algorithm, also affects the resulting difference in absolute limits. Table 3

shows how the tightness of limits changes as the number of messages passed changes. To test how the number of messages passed influences accuracy, the message passing rate was varied from 1 to 100 messages per second; the length of the experiment was held at 7.5 min; and the length of the message-passing intervals was fixed at 2.5 min.

Table 3: Difference in Limits versus Messages Passed per Second

Messages Passed per Second	Time to Calculate Limits (sec)	Difference in α_{min} and α_{max} (μs)	Difference in β_{min} and β_{max} ($x10^{-7}$)
1	1.12	234.81	6.247
20	32.23	196.49	5.176
50	77.25	195.46	5.157
100	127.48	191.58	5.046

The greater the number of messages that are passed, the smaller the difference in the limits of α and β. However, more messages will result in more points in the convex hull. This increases the memory requirements for an experiment, as well as the time it takes for the limits on α and β to be computed. As can be seen, the eventual growth of the time to calculate limits is sublinear with respect to the number of data points. This growth in time to calculate values correlates directly to the number of points in the convex hull. As more data points are taken, the number of vertices of the hull grows.

These results show that the longer the interval from passing the first message before the experiment to passing the last message after the experiment, the better the limits will be. The difference in β_{max} and β_{min} is in inverse linear proportion to the time between passing the first message and receiving the last one. The difference between α_{max} and α_{min} also decreases as time increases, but the rate of reduction is much less. The difference in limiting α and β is also reduced by passing more messages over the same amount of time. This increase in the number of data points has the effect of reducing both α and β differences, but it increases the time it takes to calculate these values.

As a general rule, passing a limited number of messages for a short period of time both before and after the experiment is a sufficient starting point for obtaining initial limiting α and β. If the precision obtained is not sufficient, greater precision in α is best obtained by passing more messages. For more precise β, the message-passing period both before and after the experiment should be extended. Both of these actions will increase the time to obtain results.

For α, the increased time is due to calculation. For β, the increased time is due to spending more time obtaining data points. These methods allow for increased precision, but are limited by the practical consideration of time constraints.

The current implementation of timeline synchronization in Loki uses one machine as a reference machine. All of the other machines have their clocks adjusted to match the reference machine. As these results show, accurate results can be obtained using just messages between the reference machine and the target machine. Although the methods using causal information and messages passed between nonreference machines could be used to lessen the bounds, this has not been found to be necessary in the test cases performed. In no case did the observations recorded have any overlapping bounds. While additional accuracy may be desirable in some circumstances, the application of causal information or messages between nonreference machines in the cases for which Loki has been used has proven unnecessary.

5. FAULT-INJECTION TESTING

5.1. Overview

When faults are injected during a campaign, they are injected based on the local node's understanding of the state of the entire system. The global view of state is updated in the local nodes by update messages passed from other nodes. Because Loki is designed to be as non-intrusive as possible, state machines do not block when transmitting state update messages. This means that there may be an inconsistent view of global states across the nodes. If a fault is injected when the views of global state are inconsistent, the fault may not have been injected in the state originally desired.

Since only experiments in which faults are injected properly are desired, fault injections are checked after the experiment is done. If the global views of state are inconsistent, the fault-injection experiment results are thrown out. Due to the measure-driven partial global view of state used by Loki, proper fault injection may only depend on some of the nodes agreeing on the global state of the system. The faults for which to check, and which nodes must be checked for their state at the time of injection, are determined by the *fault conditional* defined by the user. In that way, only the necessary nodes are checked for agreement on when and where the fault was injected.

5.2. Requirements

A fault conditional must specify what to test for proper fault injection during each experiment. The basis for a fault conditional is the *fault expression*. The fault expression is composed of several identifying characteristics of any correct fault injection. These include the *fault id*; the state in which the fault was intended to be injected; the state machine that was supposed to inject the fault; and the host on which that state machine resides. Thus, a fault expression would appear as follows:

$$fault\_id : state : mach : host$$

For ease of use, any of these fields can have an asterisk (*) as the name. This indicates that any value for that field will satisfy the requirements. This can be useful if, for instance, there are two possible faults that may be injected during a state, and the injection requirements

would be satisfied if either of them was injected. This option is provided for ease of use, as it reduces the number of fault expressions that must be listed in a fault conditional.

The fault conditional is a Boolean expression composed of fault expressions. For instance, if an experiment was supposed to inject fault *f1* in state *s1* on state machine *m1* on host *h1*, as well as inject fault *f2* in state *s2* on state machine *m2* on host *h2*, the fault conditional would be

$$(f1:s1:m1:h1)\&(f2:s2:m2:h2)$$

The fault conditional can also incorporate the | (or) operator and the ~ (not) operator. That means that it is possible to test for one of several possible faults and to test whether a fault was not injected under specified conditions.

This expression is not sufficient for the general specification of correct fault injections, however. As was mentioned earlier, one of the most important tasks in checking for proper fault injection is ensuring that the local node where the fault is injected has the proper view of the global state. That means that it is necessary to test not only whether the local state was correct where the fault was injected, but also whether other nodes were in the expected state for a proper global view. This test can be accomplished by including a *specifier* in each fault expression. There are two possible cases that can be specified. One possibility is that the fault specified is to be injected locally in the state, state machine, and host listed. The other possibility is that the node specified should be in the state listed when the fault specified is injected elsewhere in the system. If a fault *f1* is to be injected in state *s1* (on state machine *m1* on host *h1*) while state machine *m2* (on host *h2*) is in state *s2*, the fault conditional would appear as follows:

$$(f1:LOCAL:s1:m1:h1)\&(f1:NON-LOCAL:s2:m2:h2)$$

This tests not only whether the fault was injected in the proper state locally (LOCAL), but also whether the other nodes of interest were in the correct state when the fault was injected on another node (NON-LOCAL).

One question left unresolved with this fault expression is whether the fault must be injected every time the specified state is encountered, or if the intention is that it needs to be injected at least once. For instance, it may be desired that every time a state is entered, a fault is injected. On the other hand, it may be the case that a fault is injected in a state only if

certain conditions are met. There needs to be a flag that specifies the case for which to test. One option tests whether the fault is always injected, while another option only tests whether a fault is injected at least once in that state during an experiment. A flag that allows the options of ONCE or ALWAYS is included in the fault expression following the state field:

$$fault\_id : \frac{LOCAL}{NON-LOCAL} : state : \frac{ONCE}{ALWAYS} : mach : host$$

A similar question can be asked for the state machine node. Since node names are not unique, just specifying a node name does not provide a unique identifier. A unique instance number is assigned to each node with the same name. If one wishes to test for fault injection in a specific state machine, providing the node name and the instance number is sufficient to identify a specific state machine. However, one may also wish to test for fault injection in all nodes of a given name, or a certain number of nodes of that name, or even greater or less than a certain number of nodes of that name. Such a need may arise when testing an election protocol, for instance. One may wish to inject a fault into a certain number or percentage of nodes to test the results from the election protocol. To provide that capability, there need to be two additional fields in the fault expression: one that identifies a number (NUMBER) and one that identifies what that number stands for (NUMBER_MODE). Table 4 lists the options.

Table 4: Identifier Descriptions

NUMBER_MODE	NUMBER Description
INSTANCE_NUM	Instance number associated with state machine.
EQUAL_TO	State machines in which specified fault must be injected.
LESS_THAN	Maximum number of state machines in which fault can be injected.
GREATER_THAN	Minimum number of state machines in which fault can be injected.
ALL	All state machines of the given name must have faults injected in them; has no associated number value.

These added fields now mean that each fault expression must take the following form:

$$fault\_id : \frac{LOCAL}{NON-LOCAL} : state : \frac{ONCE}{ALWAYS} : mach : NUMBER\_MODE : NUMBER : host$$

This fault expression provides a general language that can be used to specify a variety of fault-injection possibilities. While the language provides many capabilities, most fault

conditionals will probably be simple cases of testing for one fault and the states of other nodes when it occurs. The complexity of the fault conditional will vary based on how the experiment and state machines are designed. If the state machines can be designed in such a way as to limit when and where faults are injected, the fault conditional can be expressed in simpler terms. For instance, a state machine could be designed to inject faults while in one of two states. In that case, two fault expressions would have to be tested in the fault conditional. If the state machine were changed so that the fault was instead injected from only one state, it would simplify the fault conditional and make fault-injection testing quicker and easier.

5.3. Analyzing Experiment Data

The truth of the fault conditionals is determined for each experiment by determining the truth of each of the fault expressions and then using a Boolean logic evaluator to determine the truth of the fault conditional expression. Section 5.3.1 explains how a fault expression is compared to the results to determine whether the results of an experiment match the intended outcome. Section 5.3.2 describes how the truth of the fault conditional is determined, based on the results of the fault expression evaluation.

5.3.1. Testing fault expressions

The first step in testing a fault expression is to gather information based on the state, state machine, and host given. The checker goes through the data from the Loki runtime and records information based on the fault expression. This information includes the times when each state of interest was entered and exited, and a record of whether a fault was injected locally while in that state. This information is associated with the name of the state, the state machine, and the instance number for every state of interest.

For instance, if specific names are given for the state, state machine, and host, the fault-injection tester will only record information when in that state, on that state machine, and on that host. Asterisks can be used in place of any of these names to make the search more general. For instance, if an asterisk is the name of the host, the fault-injection tester will search every host for that state and state machine. Caution must be used, however. If an asterisk is used for the state name, information about every state entered and exited on a node will be recorded. If an asterisk is used for the state name as well as the state machine name

and/or the host name, the amount of information collected can be large, especially for complex systems that change state often, or for experiments of long duration. The amount of memory used to store all the necessary information in these cases will be large. Caution should be used when writing fault expressions and when designing experiments.

The check will record all local fault injections and the states and nodes in which they were injected. If the NON-LOCAL option is chosen, the fault-injection tester will look for all remote fault injections to determine whether a fault was injected remotely while in the local state. For each fault injected, the injection time is noted. The checker then searches the list of relevant states. Figure 19 shows possible fault-injection times with respect to the state in which the fault was to be inserted. If the fault was injected after the start of the state and before the end of the state, this is noted for that state. The upper bound of the state start time and lower bound of the fault-injection time are used to determine whether the fault was injected after the state was entered. The lower bound of the state end time and upper bound of the fault-injection time are used to determine if the fault was injected before the state was exited. Using these bounds for comparison guarantees that it can be determined whether the fault was injected during the proper state. If Loki is unsure whether the fault was injected during a certain state, it conservatively assumes that it was not. Among the cases shown in Figure 19, Loki considers only case 5 a valid fault injection.

Once it is known whether faults were injected properly, both locally and remotely, for each occurrence of relevant states, it can be determined whether the requirements of the other flags specified were met. The first test is based on the ONCE/ALWAYS flag associated with the state. This flag is ONCE if it is necessary for the fault to be injected in that state at least once. The flag is ALWAYS if the fault should be injected every time that state occurs. The information for each state is checked, and if at least one of them indicates a fault injection, the ONCE flag is satisfied. If all of the states indicate a fault injection, the ALWAYS flag is satisfied.

The second flag that must be checked is the NUMBER_MODE flag, with its associated NUMBER. If the NUMBER_MODE is INSTANCE_NUM, the state on the state machine with instance number NUMBER is checked for compliance. If the

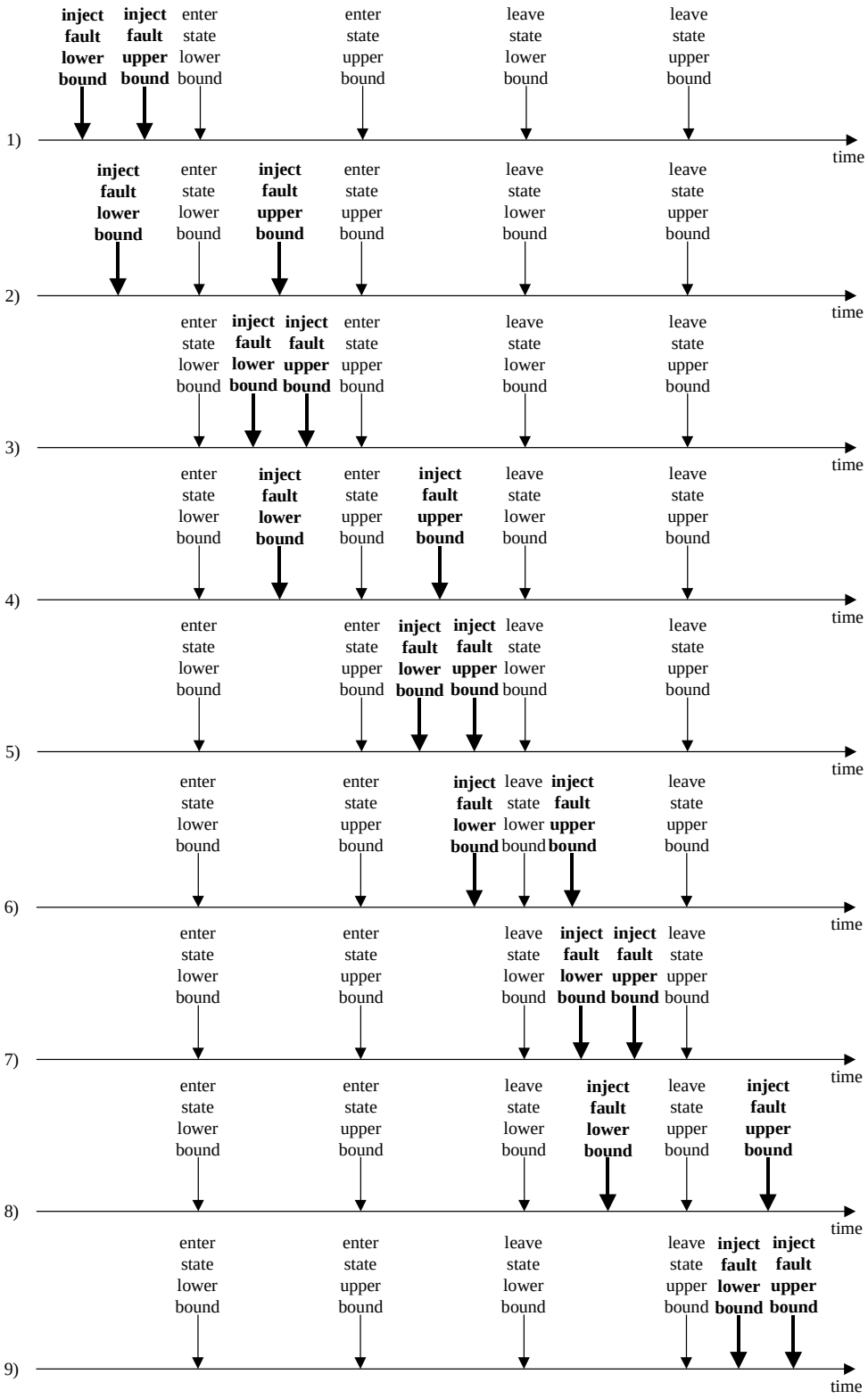

Figure 19: Timeline of Injection of a Fault in a State

NUMBER_MODE is ALL, every state machine must have the fault injected properly. If the NUMBER_MODE is GREATER_THAN, LESS_THAN, or EQUAL_TO, the states are checked with the proper comparator to determine whether the proper number of state machines had the fault injected in the correct state.

Once these checks are done to determine the truth-value of each fault expression, the entire fault conditional is tested for truth.

5.3.2. Evaluating a fault conditional

Once the truth of all the fault expressions is determined, the fault conditional is evaluated. This is done with a Boolean evaluator. The fault conditional is expressed in terms of fault expressions connected by ands (&), ors (|), and nots (~). The first step in evaluating the expression is to change the truth-value of all of the fault expressions to which a ~ applies. Then the expression is evaluated completely, in order of proper precedence, until a final truth-value for the fault conditional is determined.

5.4. Output

For each experiment, if all of the fault conditionals are true, then the observations from that experiment are kept and used for statistical processing of results. Otherwise, the results are ignored. Of course, it is important to keep track of how many valid observations are kept.

The results of how many fault conditionals were valid and how many were not are provided as output from the fault-injection campaign. This informs the user of the sample size for the statistical analysis. It can also be useful in helping diagnose problems with the fault conditional specification, or with the experiment design if many or all of the faults were injected improperly for one or more fault conditionals.

5.5. Fault-Injection Testing Example

The *pyramid* program described in [10] provides an example application of the fault-injection tester. While this example does not demonstrate the use of all fault-injection testing capabilities, it does demonstrate the flexible use of the fault description language for multiple faults and testing for different cases.

The *pyramid* application is a leader election protocol based on three levels of hierarchy, as seen in Figure 20. At the first level, groups of Leaf nodes choose a leader by an arbitration process. The chosen leader of a Leaf node group passes its information up to the Leaf Manager associated with the Leaf node group. The Leaf Manager nodes arbitrate among themselves to determine a Leaf Manager leader. Finally, this leader passes the election number to the single Manager node.

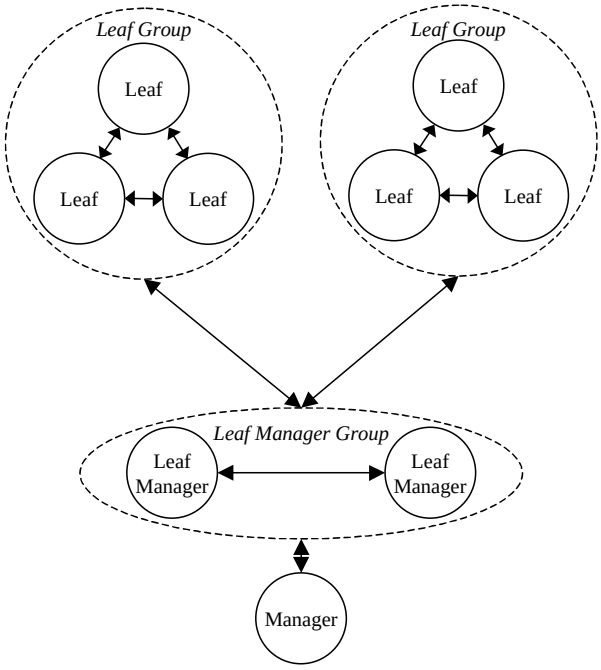

Figure 20: Configuration of Nodes for *pyramid* Application

Loki uses three different state machine types [10] to monitor the application, one for each of the different types of node. Each node has the appropriate type of state machine associated with it. The three state machines are shown in Figure 21.

Four different fault types were injected to test this example. They were given the identification numbers 10, 20, 30, and 40. Table 5 describes when and where each of the faults was injected in the experiment.

The fault with ID 10 was injected in the *Init Leaf* state, which occurs as each Leaf node is starting up. This fault was injected for each Leaf node except for instance 1. Fault 20 is injected in the *Init Leaf Manager* state of the Leaf Manager nodes. This fault was injected in each instance of Leaf Manager. The Manager node has fault 30 injected in its initialization state (*Init Manager*). Fault 40 is injected into each Leaf node that reaches the *Lead* state.

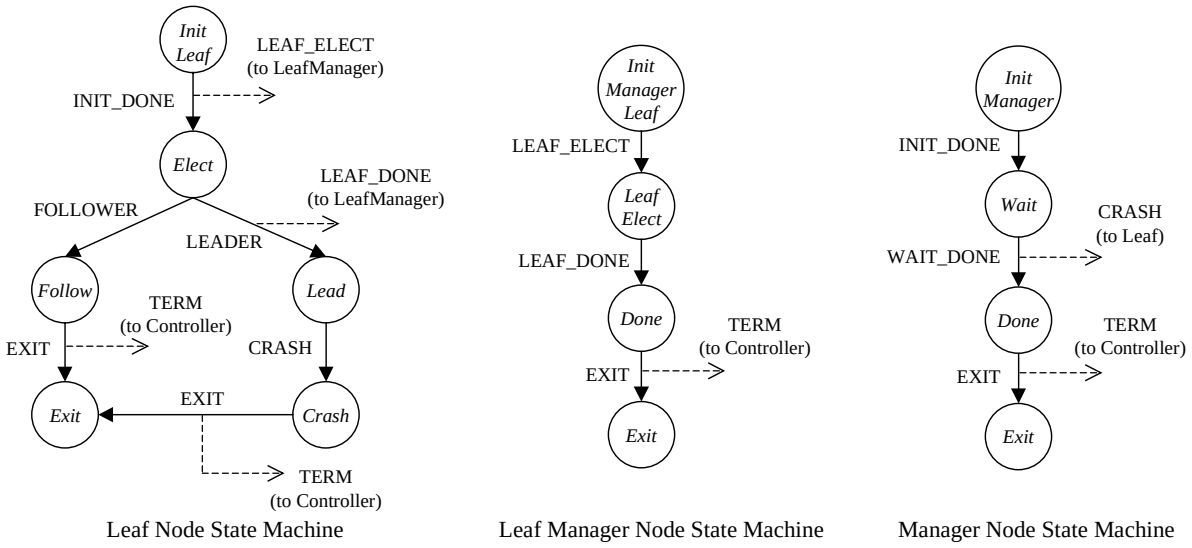

Figure 21: State Machines for *pyramid* Nodes

Table 5: Faults Injected During Experiment

Fault ID	State Injected	Node Injected	Instance	Host
10	*Init Leaf*	Leaf	2	Heathcliff
10	*Init Leaf*	Leaf	3	Heathcliff
10	*Init Leaf*	Leaf	4	Persian
20	*Init Leaf Manager*	Leaf Manager	1	Serval
20	*Init Leaf Manager*	Leaf Manager	2	Heathcliff
10	*Init Leaf*	Leaf	6	Persian
10	*Init Leaf*	Leaf	5	Serval
30	*Init Manager*	Manager	1	Persian
40	*Lead*	Leaf	3	Heathcliff
40	*Lead*	Leaf	2	Heathcliff

There are several valid ways to check that these faults were injected properly. If enough detail is known about the experiment beforehand, each fault injected can be identified separately, and then a fault conditional can be created that tests for the explicit occurrence of each fault. If desired, each fault occurrence could be listed as an individual fault conditional. In that case, each fault conditional would be tested separately, each with its own truth-value reported. Only if all of the truth-values are true will the experiment be considered successful. These examples are shown in Figure 22.

Such specific information about when the faults will be injected may not be known beforehand. Fault expressions can also be created about groups of faults. Since faults with

```
FLT_COND    (10:LOCAL:Init Leaf:ONCE:Leaf:INSTANCE:2:heathcliff)
            &(10:LOCAL:Init Leaf:ONCE:Leaf:INSTANCE:3:heathcliff)
            &(10:LOCAL:Init Leaf:ONCE:Leaf:INSTANCE:4:persian)
            &(20:LOCAL:Init Leaf Manager:ONCE:Leaf Manager:INSTANCE:1:serval)
            &(20:LOCAL:Init Leaf Manager:ONCE:Leaf Manager:INSTANCE:2:heathcliff)
            &(10:LOCAL:Init Leaf:ONCE:Leaf:INSTANCE:6:persian)
            &(10:LOCAL:Init Leaf:ONCE:Leaf:INSTANCE:5:serval)
            &(30:LOCAL:Init Manager:ONCE:Manager:INSTANCE:1:persian)
            &(40:LOCAL:Lead:ONCE:Leaf:INSTANCE:3:heathcliff)
            &(40:LOCAL:Lead:ONCE:Leaf:INSTANCE:2:heathcliff)
```

```
FLT_COND1   (10:LOCAL:Init Leaf:ONCE:Leaf:INSTANCE:2:heathcliff)
FLT_COND2   (10:LOCAL:Init Leaf:ONCE:Leaf:INSTANCE:3:heathcliff)
FLT_COND3   (10:LOCAL:Init Leaf:ONCE:Leaf:INSTANCE:4:persian)
FLT_COND4   (20:LOCAL:Init Leaf Manager:ONCE:Leaf Manager:INSTANCE:1:serval)
FLT_COND5   (20:LOCAL:Init Leaf Manager:ONCE:Leaf Manager:INSTANCE:2:heathcliff)
FLT_COND6   (10:LOCAL:Init Leaf:ONCE:Leaf:INSTANCE:6:persian)
FLT_COND7   (10:LOCAL:Init Leaf:ONCE:Leaf:INSTANCE:5:serval)
FLT_COND8   (30:LOCAL:Init Manager:ONCE:Manager:INSTANCE:1:persian)
FLT_COND9   (40:LOCAL:Lead:ONCE:Leaf:INSTANCE:3:heathcliff)
FLT_COND10  (40:LOCAL:Lead:ONCE:Leaf:INSTANCE:2:heathcliff)
```

Figure 22: Two Example Fault Conditionals

IDs 20, 30, and 40 are injected whenever a specific state is entered, fault expressions can be written that test for all occurrences of a given fault ID. Since fault 10 is not injected in this way, a different fault expression must be used. If it is desired that it be injected in exactly five initialization instances, this can be noted. How fault expressions and fault conditionals are defined depends on the exact intentions of the fault-injection campaign. Figure 23 lists two examples in which the fault conditionals will return with the result of a valid experiment.

```
FLT_COND1   (10:LOCAL:Init Leaf:ONCE:*:=:5:*)
FLT_COND2   (20:LOCAL:Init Leaf Manager:ALL:*:ALL:0:*)
FLT_COND3   (30:LOCAL:Init Manager:ONCE:*:ALL:0:*)
FLT_COND4   (40:LOCAL:Lead:ONCE:*:ALL:0:*)
```

```
FLT_COND1   (10:LOCAL:Init Leaf:ONCE:*:<:6:*)
            &(20:LOCAL:Init Leaf Manager:ALL:*:=:2:*)
            &(30:LOCAL:Init Manager:ONCE:*:<:2:persian)
            &(40:LOCAL:Lead:ONCE:Leaf:ALL:0:*)
```

Figure 23: Examples of Valid Fault Conditionals

The listing of different faults in different fault conditionals is the preferred way of describing faults. This is because a truth-value is returned for each fault conditional. By separating faults into different fault conditionals, it is easier to determine where errors

occurred when some fault is not injected properly; this can be helpful in the debugging of experiments. Each of the fault conditionals must be true for the experiment to be considered valid.

The expression below shows a fault conditional that tests for the injection of fault 10 in every initialize state for Leaf nodes.

FLT_COND1 (10:LOCAL:*Init Leaf*:ONCE:*:ALL:0:*)

The result returned would be an incorrect experiment, since the fault conditional and actual results of the experiment will be in disagreement. It is important to know what is being tested and what faults are to be injected where, so that the fault conditional can be used in properly determining correct injection. That means that the fault conditional should be as general as possible to allow testing of all faults that were injected in the proper manner, while preventing improperly injected faults from passing as properly injected.

6. SUMMARY

This thesis presents a new tool for the empirical evaluation of distributed system applications using fault injection. The main objective of the Loki empirical evaluation tool is to provide validation of systems. This is accomplished through fault injection based on a measure-driven partial global view of system state and statistically significant interpretation of experiment results to yield coverage and performance-related measures. The interpretation of results is accomplished, in part, by synchronization of local timestamps to form a single timeline with minimal intrusion, and by postanalysis of fault-injection experiments to ensure proper fault injection. For details about how to achieve a measure-driven partial global view of system state, refer to the discussion on the implementation of the Loki run time in [10].

A proper fault-injection campaign using Loki consists of designing the user inputs (including state machine descriptions, fault descriptions, event descriptions, and measures) for the experiment desired, executing the experiments based on the globally-aware Loki runtime architecture, and determining results based on events observed. This thesis focused on the interpretation of observations for proper measure estimation. This includes the accurate and nonintrusive timestamping of events, adjustment of all local clock readings to a single global timeline, and the checking of proper fault injection in postanalysis.

The timestamp synchronization is done without perturbing the system under study. Synchronization methods are used to adjust local clock readings after the experiment has completed. This allows the passing of messages before and after the fault-injection campaign, so the global clock can have sufficient accuracy without any perturbation of the system under study. Timestamps are adjusted based on a convex hull method. This produces absolute physical bounds for each adjusted timestamp. Using the fact that clock offsets are linear in nature further refines the bounds from the convex hull method. The use of assembly instructions provides the most accurate, least intrusive solution possible in software. The use of `gettimeofday()` provides a less accurate but more portable solution for Unix-like systems.

Because faults are injected during experiments without a guarantee of proper injection, a postanalysis must be performed to ensure that the faults were injected during the intended

system state. The user provides the intended partial global view of state in which a fault was to be injected in the form of a fault conditional. This fault conditional is compared against the situation in which the fault was actually injected in each experiment. Faults are checked for proper injection during the state specified. This is done for both faults that were to be injected locally and faults that were to be injected remotely while in the local state. If the fault injections are deemed successful, the results are kept and used for statistical estimation. If experiments did not have proper fault injection, the observations are not used. The fault injection specification language allows for compact representation of faults in checking for classes of faults. Fault-injection testing not only determines when faults are not injected as desired due to the nonblocking nature of state machines, but also can be useful in debugging incorrectly designed fault-injection experiments.

Remaining work for the Loki empirical evaluation tool includes integration of the postanalysis work presented in this thesis with the measure estimation tool to allow for calculation of user-defined measures based on experiment results. One possible extension of this work is the incorporation of causal information in the refinement of timestamp bounds. Another possible extension is the integration of an extensible fault conditional language. Such a language would provide the user with the ability to refine the fault conditional language to meet the specific fault-injection testing requirements for a given experiment. Also, the incorporation of on-chip timers for other processors would enable more accurate results across a wider range of platforms.

APPENDIX A: MULTIPLICATION ALGORITHM

Figure 24 shows a flowchart of the multiplication algorithm from [24].

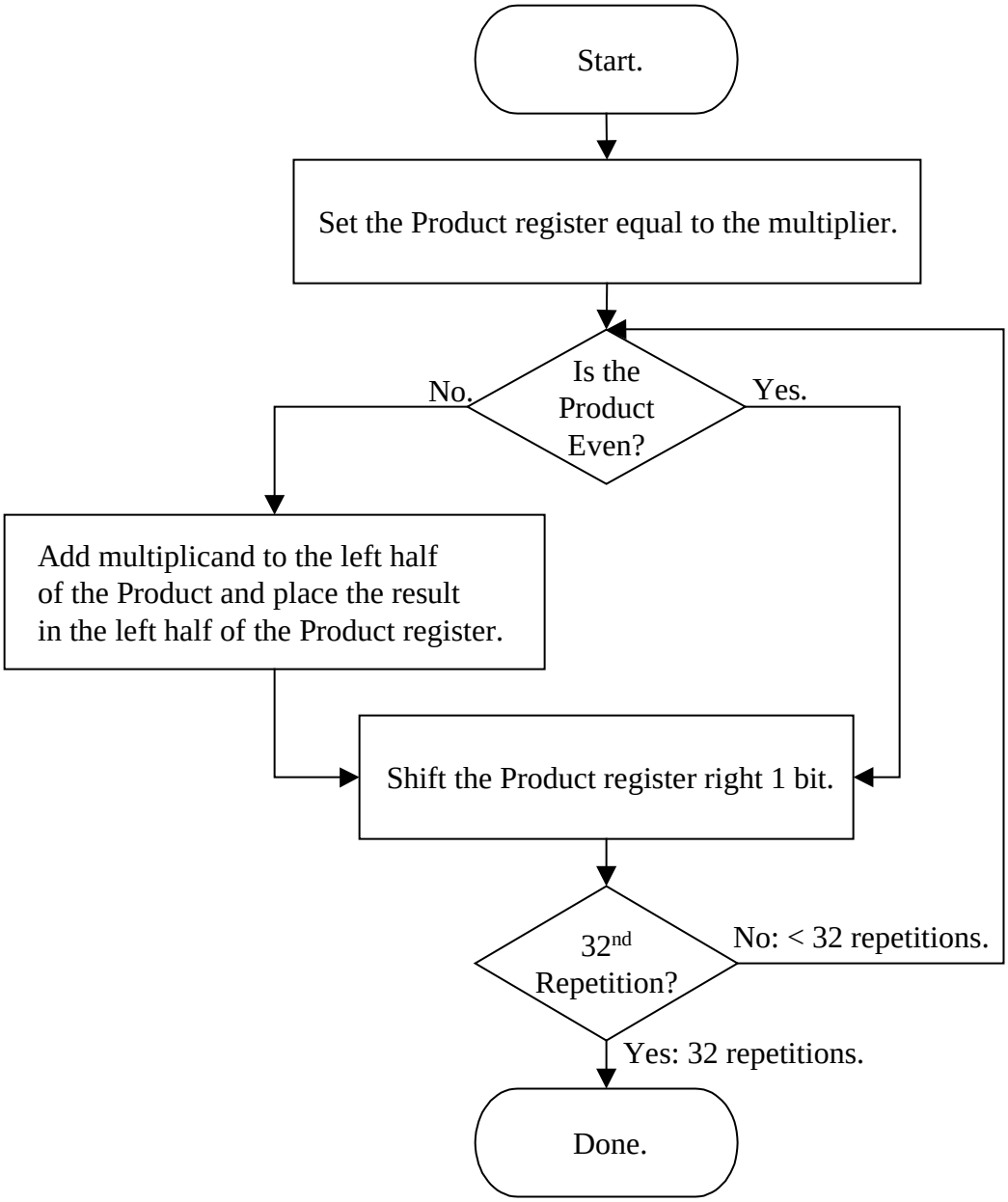

Figure 24: Flowchart of the Multiplication Algorithm from [24].

APPENDIX B: DIVISION ALGORITHM

Figure 25 shows a flowchart of the division algorithm from [24].

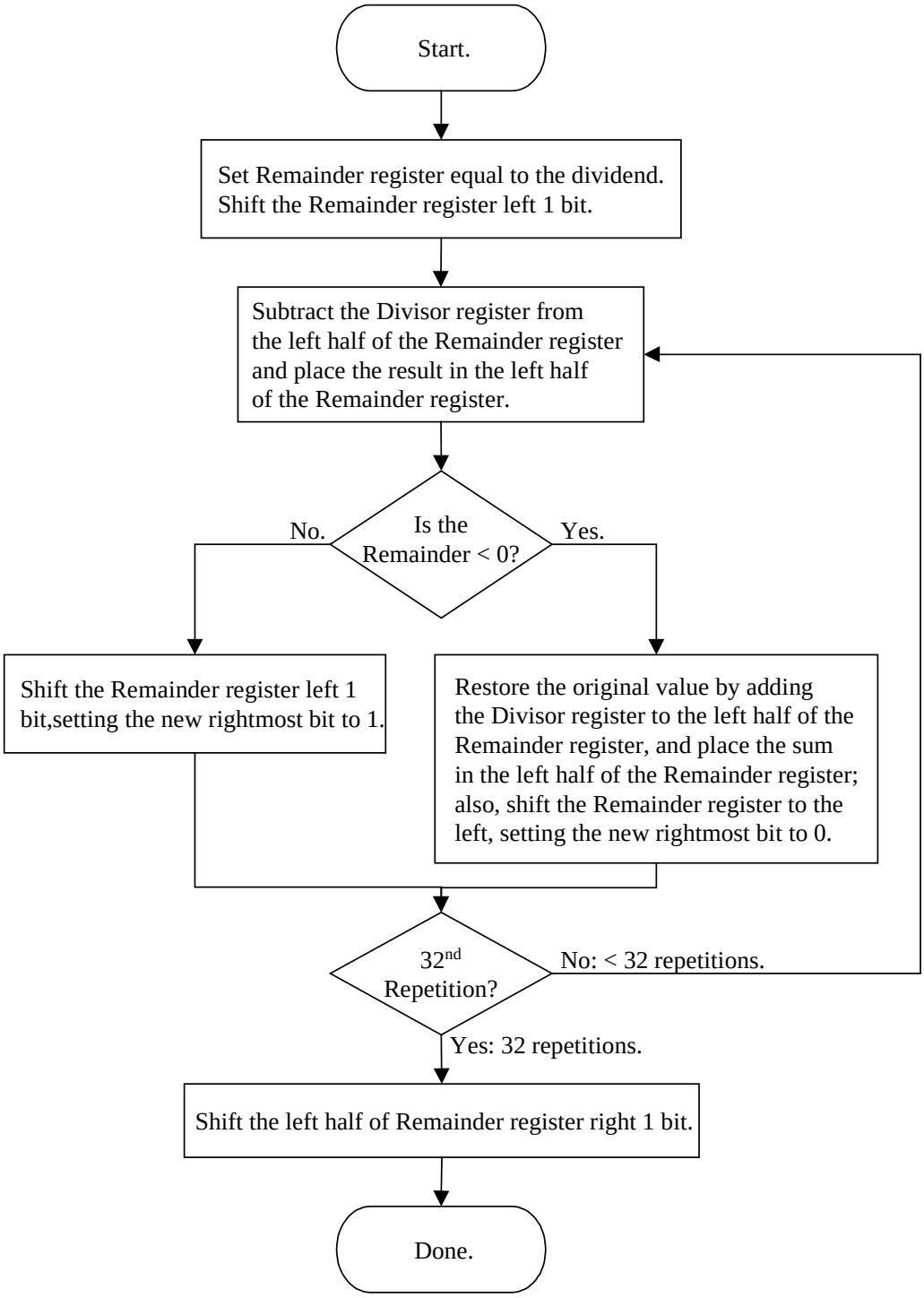

Figure 25: Flowchart of the Division Algorithm from [24]

REFERENCES

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

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

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

[4] F. Lange, R. Kroeger, and M. Gergeleit, "JEWEL: Design and implementation of a distributed measurement system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 657-671, November 1992.

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

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

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

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

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

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

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

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

[13] U. Kleinhans, J. Kaiser, and K. Czaja, “Spearmints: Hardware support for performance measurements in distributed systems,” *Micro*, vol. 13, pp. 69-78, October 1993.

[14] P. Verissimo, L. Rodrigues, and A. Casimiro, “CesiumSpray: A precise and accurate global time service for large-scale systems,” *Real-Time Systems*, vol. 12, pp. 243-294, May 1997.

[15] H. Kopetz and W. Ochsenreiter, “Clock synchronization in distributed real-time systems,” *IEEE Transactions on Computers*, vol. C-36, pp. 933-940, August 1987.

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

[17] F. Mattern, “Virtual time and global states of distributed systems,” in *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, 1989, pp. 215-226.

[18] F. Cristian, “Probabilistic clock synchronization,” *Distributed Computing*, vol. 3, pp. 146-158, 1989.

[19] L. Lamport and P. M. Melliar-Smith, “Synchronizing clocks in the presence of faults,” *Journal of the ACM*, vol. 32, pp. 52-78, January 1985.

[20] A. Duda, G. Harrus, Y. Haddad, and G. Bernard, “Estimating global time in distributed systems,” in *Proceedings of the 7th International Conference on Distributed Computing Systems*, 1987, pp. 299-306.

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

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

[23] Intel Corporation, “Using the RDTSC instruction for performance monitoring,” July 1998, http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM.

[24] D. A. Patterson and J. L. Hennesy, *Computer Organization and Design: The Hardware/Software Interface.* San Francisco: Morgan Kaufmann, 1994.

[25] A. Allen, *Probability, Statistics, and Queueing Theory with Computer Science Applications*. New York: Academic Press, 1990.

[26] The Numerical Algorithms Group, Inc., *C Library Mark 4 Manual*. Oxford,UK: The Numerical Algorithms Group, Inc., 1996.

[27] R. Sedgewick, *Algorithms*. Reading, MA: Addison-Wesley, 1988.