Plausible Clocks with Bounded Inaccuracy

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Master of Science in the

Graduate School of The Ohio State University

By

Brad T. Moore, B.S.

* * * * *

The Ohio State University

2005

Master's Examination Committee:

Paolo A. G. Sivilotti, Adviser

Bruce Weide

Approved by

Adviser

Department of Computer
and Information Science

ABSTRACT

In a distributed system, *logical time*, as opposed to physical time, is used to order the events of a computation. It orders events by their potential causal relationship: whether one event may have affected the other. *Logical clocks*, or time stamping schemes, are tools for determining the causal order between events. They store time stamps with each event, and append time tags to each message of the computation. For an asynchronous system of N processes, the time stamps and time tags of a logical clock must be of size $O(N)$ in order for the logical clock to be completely accurate. Typically, this size of $O(N)$ limits the scalability of fully accurate logical clocks (such as vector clocks).

Plausible clocks are a class of logical clocks that have a smaller size but may be inaccurate. To date, the time stamps and time tags of plausible clocks have had a fixed-size, while the clock's inaccuracy varies from run to run. In this thesis, we define a new metric, *imprecision*, that formally characterizes the fidelity of a plausible clock. We present a new plausible clock that guarantees an arbitrary bound on imprecision. This bound is maintained by allowing the size of the time tags to grow and shrink over the course of the computation. We prove this clock's correctness, present results of a simulation study, and evaluate its performance.

ACKNOWLEDGMENTS

Foremost, I would like to thank my advisor Paul Sivilotti. His insight and attention to detail were invaluable throughout this work. Also, his high standards made this work both exciting and rewarding - and forced me to push myself beyond anything I have previously done.

Bruce Weide, my other committee member, was also influential during the creation of this thesis. During my undergraduate study, I was a member of his undergraduate research group, EUROPA. My time with that group helped confirm my decision to do graduate work.

This past year I had the pleasure of being Sandy Mamrak's last graduate student before she retired. Working for her was not only enjoyable, but it also helped meet a lot of the friends I have here.

Speaking of friends, I couldn't have done this without them. Mike, Scott, Guadalupe, Greg - you all made DL674 too much fun. Bill and John - you both helped me keep my sanity, and my humility, back at our apartment.

And finally, I thank Megan. She wasn't here with me at the end, but she was for the beginning. And that I won't forget.

VITA

May 29, 1983 ..... Born - Milwaukee, Wisconsin

2004 ..... B.S.
Computer and Information Science,
The Ohio State University,
Columbus Ohio

FIELDS OF STUDY

Major Field: Computer and Information Science

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure **Page**

CHAPTER 1

Introduction

1.1 Causality

Today, distributed and parallel applications are pervasive. Despite their ubiquity, designing, implementing, reasoning about, and testing these systems remains a challenge. A primary source of difficulty is the concurrency and resulting nondeterminism inherent in distributed programs. In sequential systems, the execution of events is totally ordered. However, in distributed systems, events occurring on separate processes are both temporally and *spatially* separated. Actions in the system no longer form a totally ordered execution, but rather a partial order. Therefore physical time, which itself is totally ordered, is inappropriate for characterizing and reasoning about the behavior of a distributed application.

Logical time [9] was introduced to aid in the reasoning and construction of distributed systems. It is defined by the *happens before* $\rightarrow$ relation which orders events by potential causality: whether one event may have affected the other. The ability to determine the potential causal relationships between events is fundamental to a variety of distributed applications. For example, a global snapshot consists of a set of events such that no pair is causally related [3, 12, 5]. Cache-coherence protocols can maintain consistency by ordering distributed updates to a shared object by potential

Figure 1.1: Happens Before in a Sequential System

causality [1, 13, 6]. Resource allocation algorithms can use this relation to resolve contention for a shared resource [14, 10].

In our model of a distributed system, events may represent any action of interest. An event could be something as small as an individual machine instruction, or something as large as a database transaction. A key point is that the details of an event are abstracted away. Without the details of what an event really is, let us consider the criteria for a potential causal relationship between events. Figure 1.1 depicts the execution of a sequential system. The directed line represents the single thread of execution, and its direction denotes the execution order. Events of the system are represented by circles on the directed line. For an event e (shown in the figure), the criteria for a potential causal relationship is simple: an event happens before e if and only if it occurs before e in the execution order. Likewise, e happens before all events that occur later in the system. Therefore, in Figure 1.1, all events occurring to the left of e happen before e, and e happens before all events to the right of it.

The criteria for a potential causal relationship in the case of distributed systems is a little more complicated. Not only does a distributed system have several processes executing events, but these processes can communicate to each other via message-passing. Therefore, potential causal relationships exist between local events of a

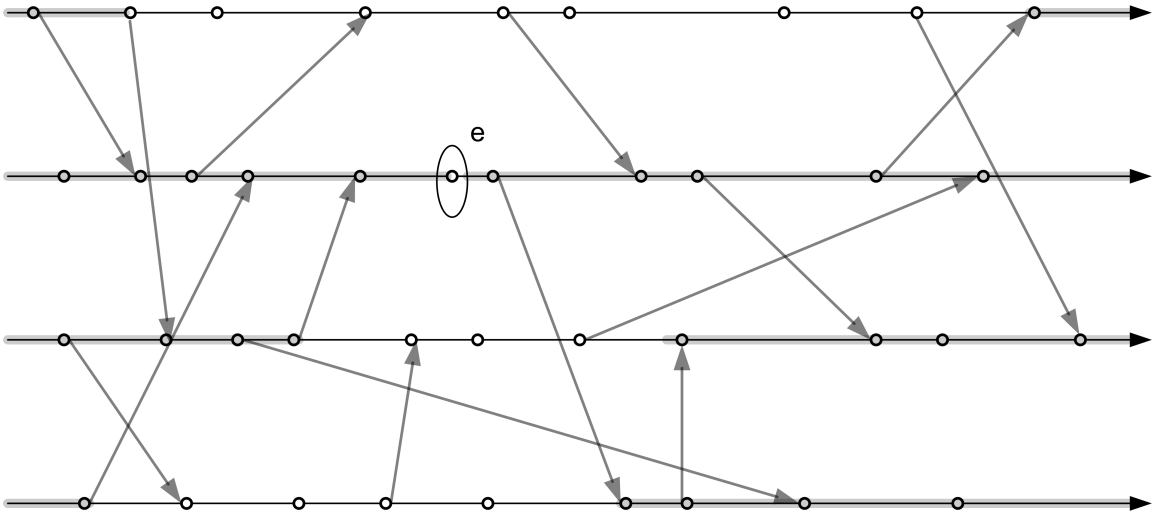


Figure 1.2: Happens Before in a Distributed System

process, *and* they exist between send-receive event pairs. Figure 1.1 depicts an execution of a distributed system. For an event e (shown in the figure), the events to the left of e that are marked by a gray line happen before it. Likewise, e happens before the grayed events to the right of e. Note that the happens before relationship is transitive.

Another important relationship between events is the *lack* of a causal relationship. When neither event happens before each other, we call these events *concurrent*. In Figure 1.1, the un-grayed events between the dashed lines are concurrent with e.

1.2 Time Stamping Schemes

A Time Stamping Scheme (TSS) [16] is a tool used by distributed applications to record logical time. That is, a TSS stores local information, called *time stamps*, with each event. It also appends information, called *time tags*, to each application message.

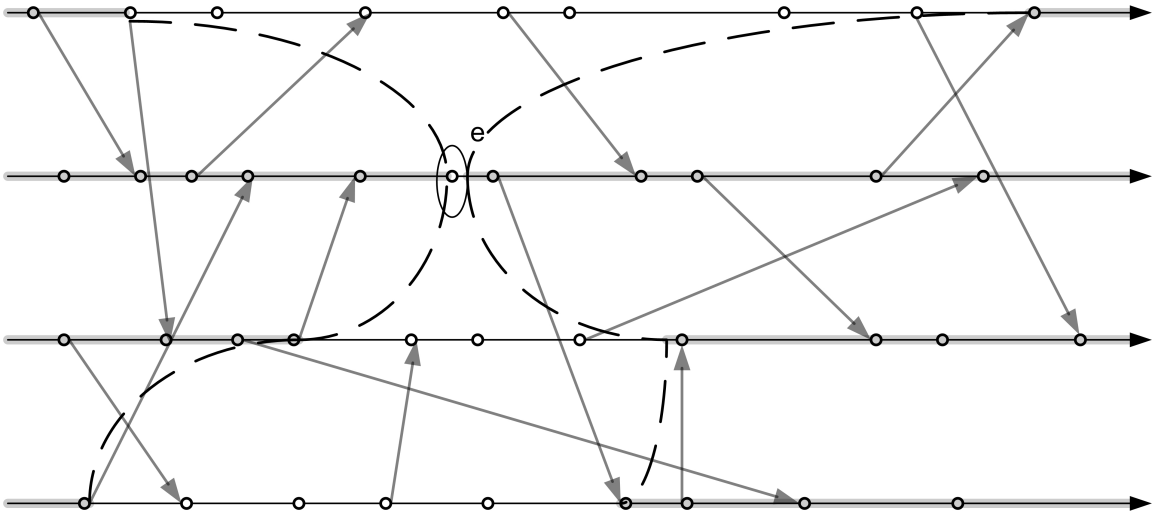


Figure 1.3: Concurrency in a Distributed System

At run-time, an application can compare two time stamps and determine the causal order of their respective events. Note, a TSS does not add events or messages to the system. Therefore, its performance is evaluated by the size of its time stamps and time tags, the time it takes to create time stamps and time tags, and the time it takes to compare two time stamps.

Two examples of TSSs are the vector clock [7, 11], and Lamport's clock [9]. For a distributed system of N processes, the time stamps and time tags of the vector clock are vectors of N integers. The vector clock satisfies the strong clock condition. That is, the order it places on its time stamps is equivalent to the happens before order of their corresponding events. Formally, a TSS X satisfies the strong clock condition if for any two events a and b, their corresponding time stamps $X.\mathbf{stamp}(a)$ and $X.\mathbf{stamp}(b)$, and X's ordering of time stamps $\stackrel{X}{\rightarrow}$, the following holds: $a \rightarrow b \equiv X.\mathbf{stamp}(a) \stackrel{X}{\rightarrow} X.\mathbf{stamp}(b)$. For a general distributed system, time stamps and

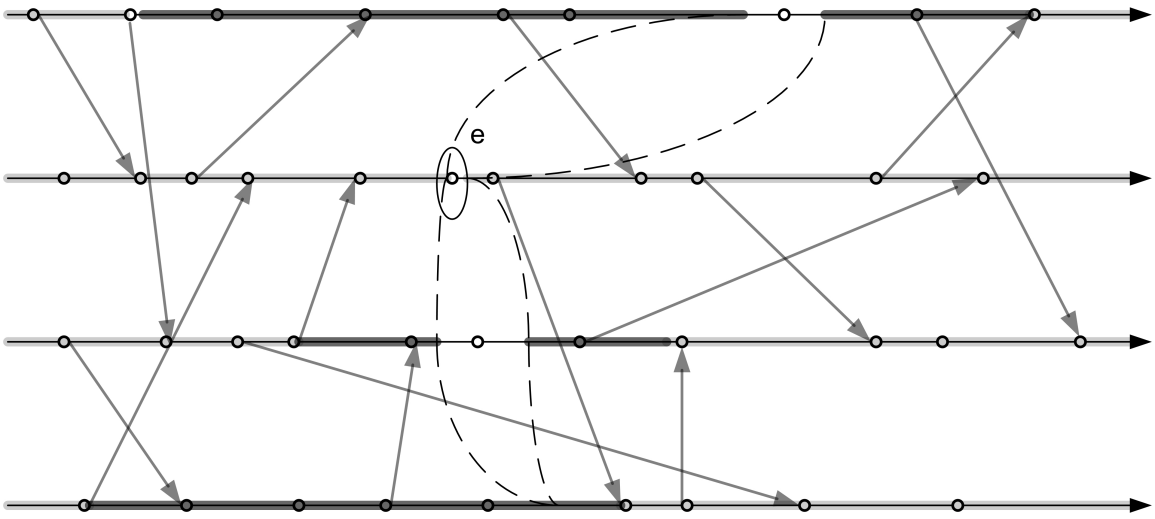


Figure 1.4: Concurrency as seen by Lamport's Clock

time tags of size $O(N)$ are required for a TSS to satisfy the strong clock condition [4]. Unfortunately, this linear size of time tags limits the scalability of such TSSs.

The time stamps and time tags of Lamport's clock are single integers. Lamport's clock satisfies the weak clock condition. That is, Lamport's clock correctly orders all causally-related events, but may incorrectly order concurrent events. Formally, a TSS X satisfies the weak clock condition if for any two events a and b, the following holds: $a \rightarrow b \Rightarrow X.\mathbf{stamp}(a) \stackrel{X}{\rightarrow} X.\mathbf{stamp}(b)$. Consider our previous example of concurrency in a distributed system (Figure 1.1). By incorrectly ordering concurrent event pairs, Lamport's clock presents a less concurrent view of the system. Figure 1.2 depicts the view of the distributed system as seen by Lamport's clock. The events marked by the dark gray line are those incorrectly ordered by Lamport's clock. The events between the dashed lines are correctly reported as concurrent with event e.

1.3 Plausible Clocks

Plausible clocks [16] were introduced as a scalable solution to vector clocks in situations where detecting concurrency impacts performance but is not necessary for correctness. A TSS X is plausible if it satisfies the weak clock condition ($a \rightarrow b \Rightarrow X.\mathbf{stamp}(a) \stackrel{X}{\rightarrow} X.\mathbf{stamp}(b)$). False orderings reported by the plausible clock hinder the performance of the computation, but do not undermine its correctness. For example, consider a cache consistency application that must maintain *causal consistency* [1]. Objects in the cache are associated with the time stamp of their most recent read or write. During an update of the cache, causal dependencies between new and old values represent a potential inconsistency in data. Therefore, old values that are causally dependent on incoming values should be invalidated. However, if any object's time stamp is concurrent with the incoming data, then it may be left in the cache. In this scenario, failing to establish the concurrency between events results in a decrease in performance (i.e., cache miss) but does not affect correctness.

1.4 Motivation

To date, plausible clocks have been parameterized by the size of the message overhead and have allowed the error in detecting concurrency to vary from run to run. These plausible clocks make *no guarantee* of their accuracy. In the worst case, the error of these plausible clocks grows unbounded with the number of events. Applications that use these plausible clocks gain from the decreased message size, but may suffer from the corresponding unbounded error. For instance, in the previous example of a cache consistency protocol, the number of unnecessarily invalidated objects could be quite high. The ability to bound the error of a plausible clock is useful

in applications where the accuracy of the clock affects performance as much as the message overhead.

1.5 Problem Overview

Plausible clocks are used in situations where missing some concurrency does not undermine correctness. Performance is a function of the size of the message overhead and the amount of concurrency missed. Previously proposed plausible clocks only guarantee performance with respect to message size; they do not guarantee performance with respect to the inaccuracy of the clock.

Our goal is to construct a plausible clock which is parameterized by a bound on inaccuracy. To be practical, our clock must not assume global information. It should achieve its inaccuracy bound using a minimal amount of message overhead (otherwise vector clocks would suffice). Some key design issues are: how to bound inaccuracy using local information, how to merge inaccurate data during a receive event and maintain accuracy, and how to efficiently encode time stamps and time tags.

We achieve this goal by the following. We quantify the inaccuracy of the system in terms of local error. We define the metric *imprecision* which is a history-independent bound on the error of time stamps, thereby allowing us to reason about and control the inaccuracy of the system using local information only. Finally, we develop a novel plausible clock algorithm that bounds imprecision by allowing the size of time tags to grow and shrink over the course of the computation.

1.6 Thesis Outline

This thesis is structured as follows. In Chapter 2, we define the system model, describes several common TSSs, and derive the definition of inaccuracy. In Chapter 3 we discuss our approach; introduce our metric imprecision, and introduce our plausible clock. In Chapter 4 we prove the correctness of our algorithm. In Chapter 5 we provide an experimental evaluation of our algorithm's performance with respect to two previously proposed plausible clocks. Finally, in Chapter 6 we present related work on causality-tracking, summarize our conclusions, and present possibilities for future work.

CHAPTER 2

Background and Definitions

2.1 Quantification Notation

In this thesis, we use a formal notation for quantification. A quantification has the form:

$$(\mathbb{Q} \quad i : r.i : t.i)$$

where $\mathbb{Q}$ is the operator, i is the bound variable (whose domain comes from context), $r.i$ is the range, and $t.i$ is the term. In order to be a valid quantification, the following constraints must hold:

1. The operator must be a binary, symmetric, associative operator with an identity element.

2. The range must be a predicate on the bound variable i.

3. The term must be an expression (that may contain i) and the type of this expression must be the same as the type of operands of the operator.

The predicate $r.i$ defines a set of values of i. Informally, if the set of values of i for which $r.i$ holds is $\{i_0, i_1, i_2, ..., i_K, ...\}$, and the identity element of $\mathbb{Q}$ is u, the

quantified expression expands to:

$$u \quad \mathbb{Q} \quad i_0 \quad \mathbb{Q} \quad i_1 \quad \mathbb{Q} \quad ... \quad \mathbb{Q} \quad i_K \quad \mathbb{Q} \quad ...$$

For example, consider the quantification $(\sum i : 1 \leq i \leq 5 \wedge even.i : 2i)$. This would expand to:

$$(+i : 1 \leq i \leq 5 \wedge even.i : 2i) = 0 + 2 * 2 + 2 * 4 = 12$$

Another example is the logical operator $\wedge$. Consider the quantification $(\wedge i : 1 \leq i \leq 5 : even.i)$. This would expand to:

$$\begin{aligned} (\wedge i : 1 \leq i \leq 5 : even.i) &= true \wedge even.1 \wedge even.2 \wedge even.3 \wedge even.4 \wedge even.5 \\ &= false \end{aligned}$$

Note, there are symbols traditionally used for the above quantifications. Therefore, in this thesis, we will replace the $+$ operator in the quantification with the symbol $\sum$ (similarly, $\wedge$ is replaced with $\forall$ and $\vee$ with $\exists$).

2.2 System Model and Definitions

We consider an asynchronous distributed system of N processes. Processes communicate via message-passing, which is point-to-point and fault-free. The execution of a process p_i consists of a finite set of local events denoted H_i. An event $e \in H_i$ may be a local, send, or receive event. In the system, there is a one-to-one correspondence between send and receive events. What a local event actually represents is application dependent. We denote the execution of the entire system as $H = (\cup i : 1 \leq i \leq N : H_i)$.

2.3 Happens Before

The *happens before* relation [9] represents the potential causal relationship between events. For two events $a \in H_i$ and $b \in H_j$, $a \rightarrow b$ if and only if:

1. $i = j$ and a occurs before b, or
2. a is a send event and b is the corresponding receive event, or
3. there exists an event c such that $a \rightarrow c$ and $c \rightarrow b$.

Two events are considered *concurrent* if neither happens before the other:

$$a \parallel b \quad \equiv \quad \neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$$

A Time Stamping System X [16] is formally defined as a tuple $\langle S, \overset{X}{\rightarrow}, G, X.\mathbf{stamp}, X.\mathbf{tag} \rangle$ where:

S is a set of logical time values used to locally record information (called *time stamps*),

$\overset{X}{\rightarrow}$ is an irreflexive transitive relation on time stamps,

G is a set of logical time values appended to messages to record information (called *time tags*),

$X.\mathbf{stamp}$ is the time stamping function mapping events to stamps, and

$X.\mathbf{tag}$ is the time tagging function mapping event time stamps to message time tags.

Note that, in practice, the $X.\mathbf{stamp}$ function is guaranteed to be locally computable by defining it inductively. First, time stamps are defined for all initial events. Then, a function on $S \times G$ is given that determines the time stamp of an event based upon the most recent local time stamp and the most recently received message time tag.

The relation $\overset{X}{\rightarrow}$ is irreflexive and transitive, therefore $\langle S, \overset{X}{\rightarrow} \rangle$ is a strict partial order. This strict partial order induces the following further relations, for all $r, s \in S$:

$$\begin{aligned} r \overset{X}{=} s &\equiv r = s \\ r \overset{X}{\parallel} s &\equiv \neg(r \overset{X}{\rightarrow} s) \wedge \neg(s \overset{X}{\rightarrow} r) \wedge \neg(r \overset{X}{=} s) \end{aligned}$$

For convenience, we will overload the definitions of these relations to allow them to directly compare events of H. For instance, given two events $a, b \in H$:

$$a \overset{X}{\rightarrow} b \equiv X.\mathbf{stamp}(a) \overset{X}{\rightarrow} X.\mathbf{stamp}(b)$$

In the realization of a Time Stamping System, comparison of time stamps within this strict partial order is implemented by a function $X.\mathbf{comp}$ that maps pairs of stamps to the set $\{\overset{X}{\rightarrow}, \overset{X}{\leftarrow}, \overset{X}{=}, \overset{X}{\parallel}\}$.

Also note, in the remainder of this thesis we will omit the X (so $X.\mathbf{stamp}$ would be written as simply $\mathbf{stamp}$) when the X is clear from context.

A TSS X is *plausible* if it satisfies the weak clock condition. That is, for two events a and b, $a \rightarrow b \Rightarrow a \overset{X}{\rightarrow} b$.

2.4 Logical Clocks

2.4.1 Lamport's Clock

Lamport's clock [9] is a plausible TSS with simple integer time stamps and time tags.

The definition of the Lamport clock is as follows. A time stamp $s \in S$ is an integer ($S = \mathbb{Z}$). The time tags of the system are identical to the time stamps ($G = S$). For a time stamp s on a process p_i, the **stamp** function updates s according to the following:

1. Initially, the value of s is equal to 1
2. For a local or send event, the value of s is incremented
3. For a receive event with time tag t, the max of s and t is taken and incremented

The **tag** function simply appends the time stamp of the send event to the message. For two time stamps, r from process p_i and s from process p_j, the $\overset{Lamport}{\rightarrow}$ relation is defined as:

$$r \overset{Lamport}{\rightarrow} s \quad \equiv \quad r < s$$

Figure 2.4.1 depicts a system using Lamport's clock. Events marked with a dark gray line are incorrectly ordered with event e. Observe that only the events whose time stamps are equivalent to e are marked as concurrent.

2.4.2 R-Entries Vector Clocks

The R-Entries Vector (REV) clock[16] was introduced as a constant-size plausible clock similar to vector clocks. Instead of each entry of the vector representing a single

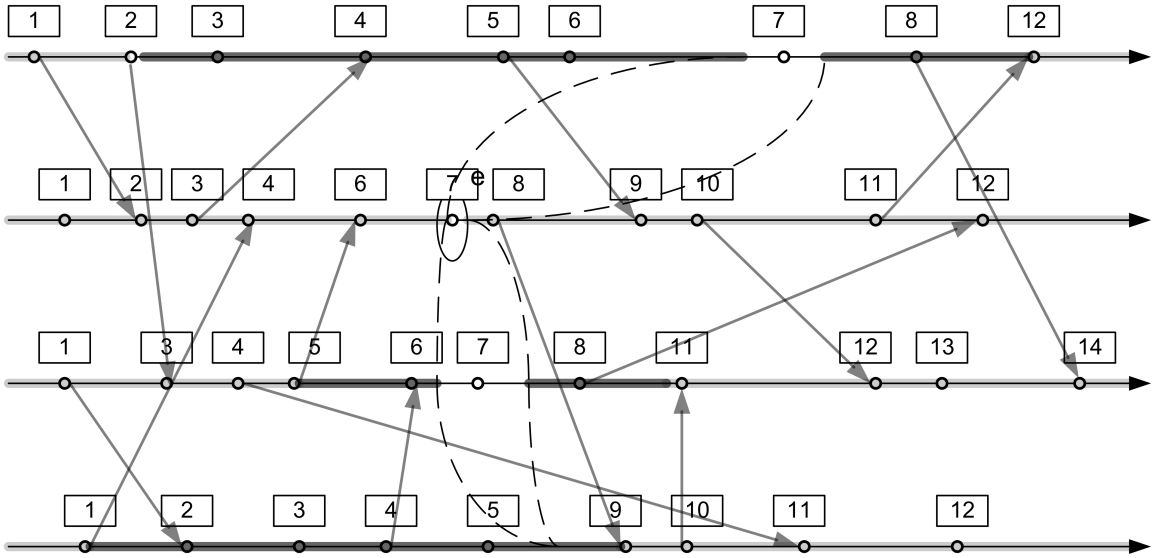


Figure 2.1: Example of Lamport's Clock

process, several processes are grouped together per entry. The function mapping a process p_i to a vector entry is $(i \ \textbf{modulo} \ R) + 1$.

The definition of the REV clock is as follows. A time stamp $s \in S$ is a vector of R entries where $R \leq N$ ($S = \mathbb{Z}^R$). The time tags of the system are identical to the time stamps ($G = S$). For a time stamp s on a process p_i, the **stamp** function updates s according to the following:

1. Initially, all entries of s are set to 0 except for $s[(i \ \textbf{modulo} \ R) + 1]$ which is set to 1

2. For a local or send event, the entry $s[(i \ \textbf{modulo} \ R) + 1]$ is incremented

3. For a receive event with time tag t, the max of each entry in t and s is taken, and the entry $s[(i \ \textbf{modulo} \ R) + 1]$ is incremented

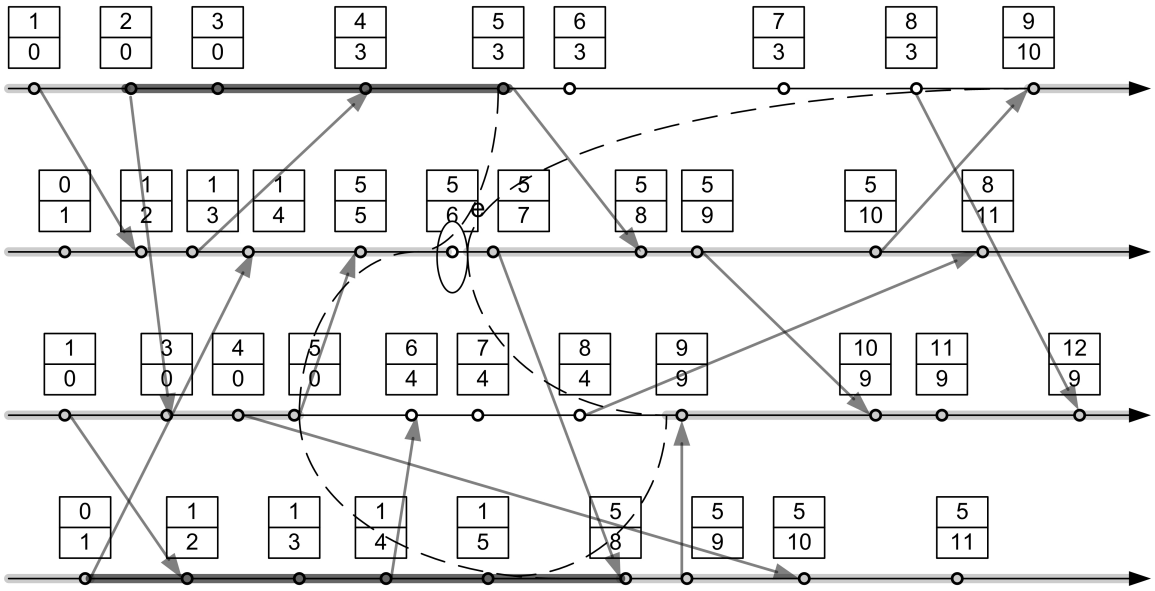


Figure 2.2: Example of REV Clock

For two time stamps, r on p_i and s on p_j, the $\overset{REV}{\rightarrow}$ relation is defined as

$$r \overset{REV}{\rightarrow} s \quad \equiv \quad (\,(\,\forall\, i \,:\, 1 \leq i \leq R \,:\, r[(i \ \textbf{modulo}\ R) + 1] \leq s[(i \ \textbf{modulo}\ R) + 1]\,) \wedge$$
$$(\,\exists\, i \,:\, 1 \leq i \leq R \,:\, r[(i \ \textbf{modulo}\ R) + 1] < s[(i \ \textbf{modulo}\ R) + 1]\,)$$

Figure 2.4.2 depicts a system using REV clock. Events marked with a dark gray line are incorrectly ordered with event e. Observe that REV incorrectly orders fewer events than Lamport's clock (Figure 2.4.1).

2.4.3 Vector Clocks

The vector clock was independently proposed by Fidge [7] and Mattern [11]. It is a TSS that satisfies the strong clock condition. That is, for two events a and b, $a \rightarrow b \ \equiv \ a \overset{Vector}{\rightarrow} b$.

The definition of the vector clock is as follows. For a distributed system of N processes, a time stamp $s \in S$ is a vector of N integers ($S = \mathbb{Z}^N$). The time tags of the system are identical to time stamps ($G = S$). For a time stamp s on a process p_i , the **stamp** function updates s according to the following:

1. Initially, all entries of s are equal to 0 except for the i^{th} entry which is equal to 1 .

2. For a local/send event, the i^{th} entry of s is incremented

3. Upon receiving a time tag g , the entry-wise max of s and g is taken and then the i^{th} entry is incremented.

The **tag** operation simply appends the time stamp of the send event to the message. The $\overset{Vector}{\rightarrow}$ relation is defined as:

$$r \overset{Vector}{\rightarrow} s \quad \equiv \quad (\forall i \,:\, 1 \leq i \leq N \,:\, r[i] \leq s[i]\ ) \wedge$$
$$(\exists i \,:\, 1 \leq i \leq N \,:\, r[i] < s[i]\ )$$

Figure 2.4.3 depicts a system using the vector clock. Observe that all events that are concurrent with event e are reported as such.

Intuitively, a vector clock maintains the following properties. First, it maps the events of a given process to a strictly increasing sequence of integers. That is, for two time stamps $r = \mathbf{stamp}(a)$ and $s = \mathbf{stamp}(b)$ on process p_i , the following holds:

$$a \rightarrow b \equiv r[i] < s[i]$$

In Figure 2.4.3, these i^{th} entries are marked with gray.

Second, it records the *most recent happens before event* from each process. For instance, consider a time stamp $r = \mathbf{stamp}(a)$ on process p_i . For each entry $r[j]$

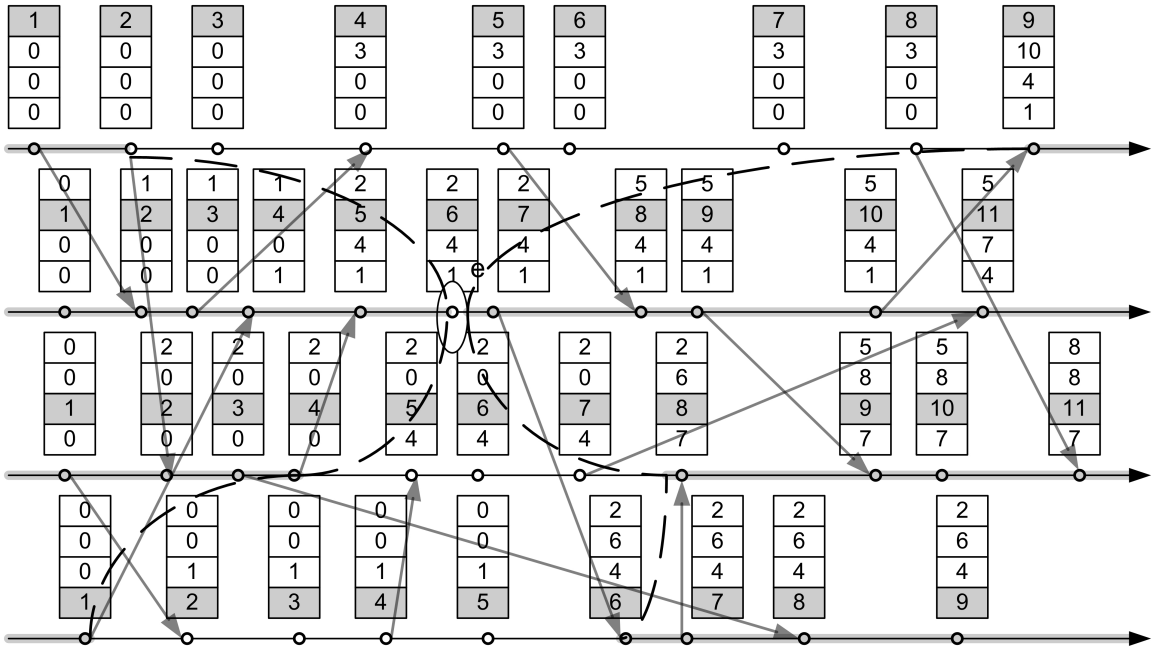


Figure 2.3: Example of Vector Clock

where $j \neq i$, there exists a time stamp $s = \mathbf{stamp}(b)$ on p_j such that $s[j] = r[j]$. This event b is the most recent event on p_j that happens before a. More formally, $b \rightarrow a \ \wedge \ \neg (\exists c \in H_j \ :: \ b \rightarrow c \ \wedge \ c \rightarrow a)$.

For example, consider event e in Figure 2.4.3. As previously mentioned, the gray entries map the local events of a process to a strictly increasing sequence of integers. For event e, the events that most recently happen before it are: event 2 from process 1, event 4 from process 3, and event 1 from process 4. Observe that the corresponding entries of e's time stamp are consequently 2, 4, and 1.

2.5 Inaccuracy

A plausible clock always correctly orders causally-related events, but it may incorrectly report a causal relationship when none exists (i.e., the events are concurrent). The *inaccuracy* of a plausible clock is the ratio between the number of incorrectly ordered event pairs, and the total number of concurrent pairs in the system [8]. Formally, we define C as the set of concurrent pairs in the system, and M as the set of incorrectly ordered pairs. Inaccuracy, $\rho(P, H)$, is therefore defined as:

$$\begin{aligned} C &= \{\, (a,b) \in H \times H \;:\; a \parallel b \;:\; (a,b) \,\} \\ M &= \{\, (a,b) \in H \times H \;:\; a \parallel b \wedge \neg(a \overset{P}{\parallel} b) \;:\; (a,b) \,\} \\ \rho(P,H) &= \frac{|M|}{|C|} \end{aligned}$$

Accuracy can then be defined as $1 - \rho(P, H)$.

Observe that $(a \parallel b \equiv b \parallel a)$ and $(a \parallel b \wedge \neg(a \overset{P}{\parallel} b) \equiv b \parallel a \wedge \neg(b \overset{P}{\parallel} a))$. Therefore, in C and M, we are counting the pair (a, b) and (b, a). This double-counting cancels itself in the above definition of inaccuracy. On the other hand, consider the following formal definition of inaccuracy from a published plausible clocks paper [8]:

$$\rho_*(P,H) = \frac{|\{\, (a,b) \in H \times H \;:\; a \parallel b \wedge a \xrightarrow{P} b \;:\; (a,b) \,\}|}{|\{\, (a,b) \in H \times H \;:\; a \parallel b \;:\; (a,b) \,\}|}$$

In the definition of ρ_*, the use of $a \xrightarrow{P} b$ instead of $\neg(a \overset{P}{\parallel} b)$ removes the double-counting in the numerator. Therefore, the function ρ_* returns half the value of the ratio between mistakes and concurrent pairs:

$$\rho_*(P,H) = 2 * \rho(P,H)$$

CHAPTER 3

Approach

With vector clocks, the time stamps of events on process p_i all differ in their i^{th} entry. This one entry orders these events and distinguishes between them. The other entries serve a different purpose: Each one uniquely identifies the most recent happens-before event on the corresponding remote process. Our approach is conceptually similar. Time stamps are vectors where the i^{th} entry orders and distinguishes between events on p_i , while the other entries indicate the most recent happens-before events on remote processes. The difference is that a *range* of values, rather than a single one, is used as an entry in the array and hence the most recent happens-before events are not uniquely identified. We will show later that using a range of values as an entry will allow us to compress the size of our time tags.

3.1 Imprecision

Our goal is to create a plausible clock that can guarantee an arbitrary bound on inaccuracy. To be practical, there should be no presumption of global information nor should the clock modify the underlying computation (*e.g.*, by sending extra messages). Our approach is to bound the inaccuracy by controlling the maximum possible error

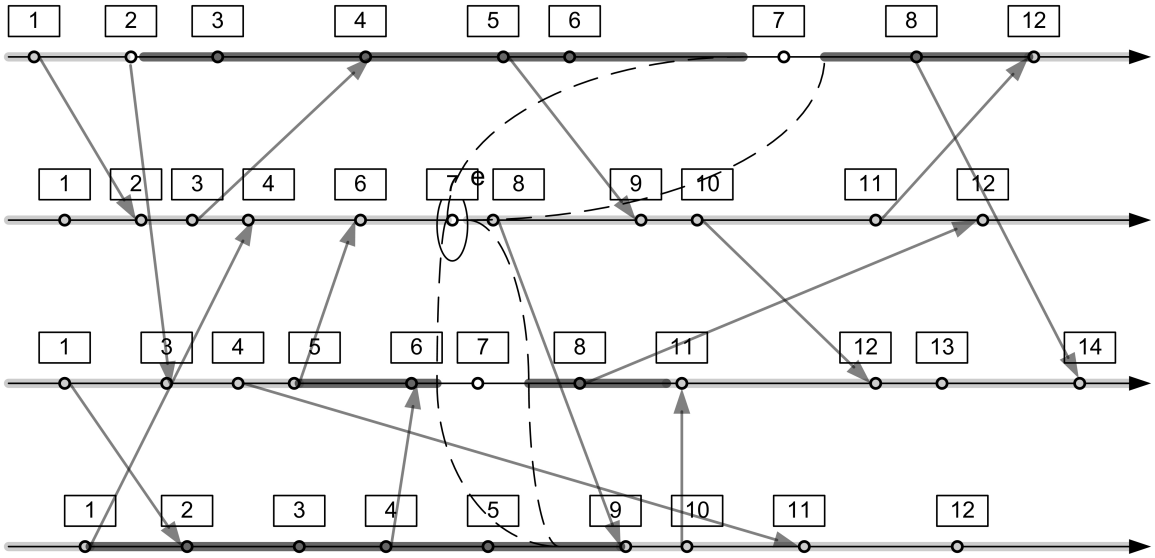


Figure 3.1: Local Error of Lamport's Clock

permitted by individual time stamps. To do so, we must first redefine inaccuracy in terms of this error.

First, we define the *local error* of a plausible clock to be the number of mistakes it makes with respect to a given event. More precisely, it is the number of (concurrent) events that are mistakenly ordered *before* the event in question. Formally, we define the local error of P with respect to an event b as:

$$\delta(P, H, b) \quad = \quad |\{\, a \in H \;:\; a \parallel b \wedge a \xrightarrow{P} b \;:\; a \,\}|$$

Figure 3.1 depicts a system using Lamport's clock. For event e, $\delta(P, H, e)$ is equal to the number of events marked by dark gray that would be ordered as happening before e. In this example, $\delta(P, H, e)$ is equal to 9 (4 events from process 1, 1 event from process 3, and 4 events from process 4).

We can now define the total number of mistakes in terms of the local error for each event. Note, that since $\delta(P, H, b)$ is defined with the $\stackrel{P}{\rightarrow}$ relation, if a pair (a, b) is counted, then we are guaranteed that the pair (b, a) will not be ($a \stackrel{P}{\rightarrow} b \Rightarrow \neg(b \stackrel{P}{\rightarrow} a)$).

$$|M| = 2 * (\sum b \in H \ :: \ \delta(P, H, b) \ )$$

Therefore, inaccuracy can be written:

$$\rho(P, H) = 2 * \frac{(\sum b \in H \ :: \ \delta(P, H, b) \ )}{|C|}$$

Our definition of $\rho(P, H)$ is still problematic. We would like to define inaccuracy in terms of the local error per event; however, it is currently the ratio between the sum of local error and the total number of concurrent event pairs. To this end, we define $\epsilon(H)$ as the ratio between the total number of concurrent pairs and the total number of events:

$$\epsilon(H) \ = \ 1/2 * \frac{|C|}{|H|}$$

For computations that exhibit regular communication patterns and whose processes are not partitioned, the value of $\epsilon(H)$ remains constant as H is extended with new events. If the processes were partitioned (say one process ceases to communicate), this ratio would increase without bound as H is extended with new events. For the remainder of the paper, we will assume fault-free executions where all processes actively communicate within the system. Rewriting the total number of concurrent pairs in terms of this concurrency ratio, we have:

$$\rho(P, H) \ = \ 1/\epsilon(H) * \frac{(\sum b \in H \ :: \ \delta(P, H, b) \ )}{|H|}$$

Since we assume that $\epsilon(H)$ is a constant, we need only to bound the mean value of δ in order to bound the inaccuracy. Unfortunately, we cannot use δ directly in

our algorithm; the **stamp** function is defined inductively over time stamps and not histories. Therefore, we define a new metric that is based on time stamps and hence can be used directly by a plausible clock to reason about fidelity. We call this metric *imprecision*. The imprecision of a time stamp generated by a plausible clock is an upper bound on the number of ordering mistakes made for an event with that time stamp. More formally, let $\mathbb{H}(P, r)$ be the set of histories for which the plausible clock P generates the time stamp r:

$$\mathbb{H}(P, r) \;=\; \{\, H \;:\; (\,\exists\, a \in H \;::\; P.\mathbf{stamp}(a) = r\,) \;:\; H \,\}$$

Imprecision, $\psi(P, r)$, is then defined as:

$$\psi(P, r) \;=\; (\,\mathbf{Max}\, H \in \mathbb{H}(P, r),\; a \in H \;:\; P.\mathbf{stamp}(a) = r \;:\; \delta(P, H, a)\,)$$

Intuitively, imprecision is the worst-case value of δ for an event with a given time stamp. Note that imprecision is independent of history and therefore is a function of the information contained within a time stamp. Consider the following example. Figure 3.1 depicts a system using Lamport's clock. Observe that every event, from other processes, that Lamport's clock reports to have happened before event e is in fact concurrent with e. Therefore, this is an example of a worst-case history for an event with time stamp 6. From this example, we can determine that the imprecision of a Lamport's time stamp of 6, in a system of 4 processes, is $5 * 3 = 15$. We can generalize this result. For a given Lamport's time stamp s, $\psi(\mathit{Lamport}, s) = (N - 1) * (s - 1)$.

We can use imprecision to bound inaccuracy. If we guarantee that all time stamps generated during a computation have an imprecision less than some arbitrary bound, K, then the mean value of δ is also less than that bound. More formally, the resulting

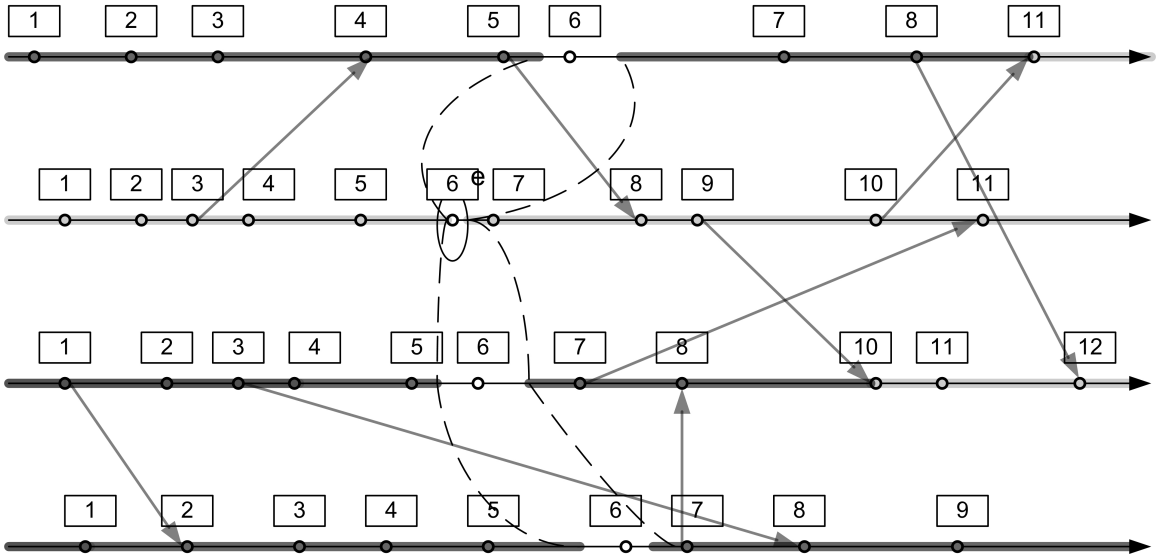


Figure 3.2: Imprecision of Lamport's Clock

bound on inaccuracy is:

$$\begin{aligned} \rho(P,H) &\leq 1/\epsilon(H) * \frac{(\sum b \in H \ :: \ \psi(P, P.\mathbf{stamp}(b)) \ )}{|H|} \\ &\leq 1/\epsilon(H) * K \end{aligned}$$

3.2 Algorithm

At the core of our algorithm is the concept of a time *interval*. A time interval is a tuple $\langle beg, end \rangle$ where beg and end are integers and $beg \leq end$. Unlike the integer entry of vector clocks which corresponds to a single event, a time interval corresponds to a set of events. The event of interest is within this range. Thus, when comparing two time intervals, we can conclude something about the ordering of the respective events of interest only when the ranges do not overlap. The ordering between two

intervals m and n is given by:

$$\begin{aligned} m \overset{interval}{<} n &\equiv m.end < n.beg \\ m \overset{interval}{\approx} n &\equiv \neg(m \overset{interval}{<} n) \wedge \neg(n \overset{interval}{<} m) \\ m \overset{interval}{\lessapprox} n &\equiv (m \overset{interval}{<} n) \vee (m \overset{interval}{\approx} n) \end{aligned}$$

We define a *precise* interval to be one in which the begin and end points are equal. In the case of precise intervals, an overlap reflects exact equality.

$$\begin{aligned} \mathbf{precise}(m) &\equiv m.beg = m.end \\ m \overset{interval}{=} n &\equiv \mathbf{precise}(m) \wedge \mathbf{precise}(n) \wedge m = n \end{aligned}$$

Since the beg and end values of a precise interval are equivalent, there is really a single value that is associated with a precise interval. For convenience, we define $m.val$ as this value for a precise interval m:

$$m.val = m.beg = m.end$$

These notions of time interval comparison are illustrated in Figure 3.2.

The time stamps of our system are analogous to those of vector clocks. A time stamp $s \in S$ is a vector of N time intervals. Like vector clocks, the time stamps of our system map the events of a given process to an increasing sequence. That is, for two time stamps $r = \mathbf{stamp}(a)$ and $s = \mathbf{stamp}(b)$ on process p_i, the following hold:

$$\mathbf{precise}(r[i]) \wedge \mathbf{precise}(s[i])$$

$$a \rightarrow b \equiv r[i] \overset{interval}{<} s[i]$$

For a given process p_i, the integer entries of vector clocks (excluding the i^{th}) are used to indicate the most recent happens before events on remote processes. Formally

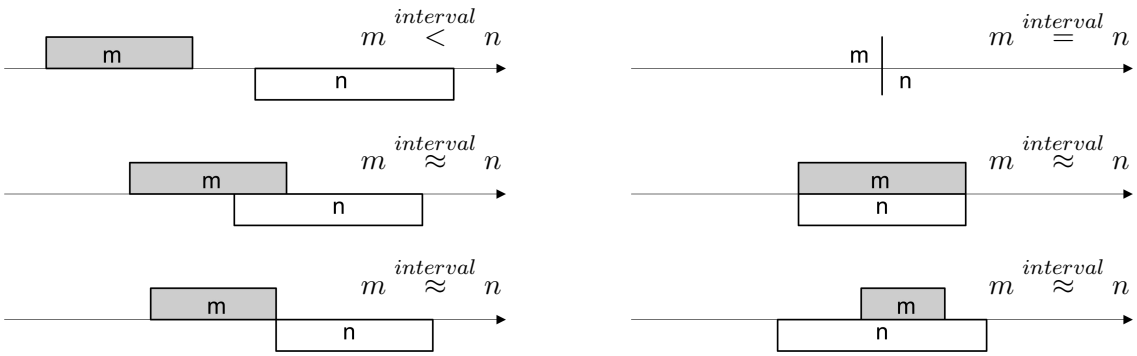


Figure 3.3: Time Interval Comparison

we define the most recent event on p_j that happens before an event a as the (unique) event $\mathbf{mrhb}(a, j)$ that satisfies the following:

$$\mathbf{mrhb}(a, j) \in H_j \wedge$$

$$\mathbf{mrhb}(a, j) \rightarrow a \wedge$$

$$\neg (\exists c \,:\, c \in H_j \,:\, \mathbf{mrhb}(a, j) \rightarrow c \wedge c \rightarrow a \,)$$

While vector clocks use integer entries to precisely indicate single events, we use time intervals to represent a *range* of events. We guarantee that the event of interest, the most recent happens before event, is within this range. Formally, for an event a on process p_i with time stamp $r = \mathbf{stamp}(a)$:

$$(\forall j \,:\, 1 \leq j \leq N \wedge j \neq i \,:\, r[j].beg \leq \mathbf{stamp}(\mathbf{mrhb}(a, j))[j].val \leq r[j].end \,)$$

A time stamp s also satisfies several additional properties. First, all imprecise intervals of s share the same end value. Second, all precise intervals of s are greater

than the imprecise intervals. Formally, these invariants are:

$$(\exists k :: (\forall i : 1 \leq i \leq N : \neg\textbf{precise}(s[i]) \Rightarrow s[i].end = k))$$

$$(\forall i, j : 1 \leq i, j \leq N : \neg\textbf{precise}(s[i]) \wedge \textbf{precise}(s[j]) \Rightarrow s[i].end \leq s[j].val)$$

Note, since precise intervals can be encoded with a single integer and all imprecise intervals share the same end point, the time stamps of our system can be encoded with $N + 1$ integers. Therefore, assuming integers require L bits, a time stamp requires $(N + 1) * L$ bits.

A time tag t is also a vector of N time intervals. It satisfies all the properties of time stamps and, in addition, the property that imprecise intervals have the same begin point.

$$(\exists k :: (\forall i : 1 \leq i \leq N : \neg\textbf{precise}(s[i]) \Rightarrow s[i].beg = k))$$

Figure 3.2 presents an example of a time stamp and a time tag.

Given a time tag t with R precise intervals, it requires $(\log N) * R$ bits to encode the association between precise intervals and their respective processes. Assume integers require L bits. Since all imprecise intervals are the same, a time tag requires $R * (L + \log N) + 2L$ bits.

3.2.1 comp()

The comparison of time stamps in our algorithm is similar to that of vector clocks. The $\xrightarrow{P}$ relation, and thereby the **comp** function, is formally defined as:

$$\begin{aligned} r \xrightarrow{P} s \quad \equiv \quad & (\forall i : 1 \leq i \leq N : r[i] \overset{interval}{\lessapprox} s[i]) \wedge \\ & (\exists i : 1 \leq i \leq N : r[i] \overset{interval}{<} s[i]) \end{aligned}$$

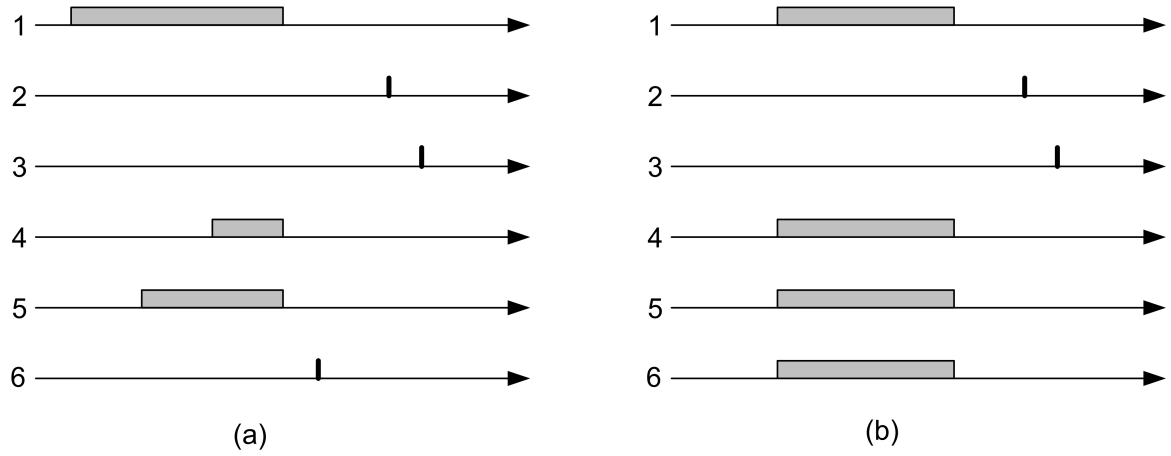


Figure 3.4: Example of a time stamp and a time tag

Data: r is old stamp on p_i, s is new stamp on p_i, t is the incoming tag
INITIALLY:
for $j := 1$ ***to*** N **do**
$s[j] := \langle 0, 0 \rangle$
end
$s[i] := \langle 1, 1 \rangle$
LOCAL or SEND EVENT:
for $j := 1$ ***to*** N **do**
$s[j] := r[j]$
end
$s[i].end := s[i].end + 1$
$s[i].beg := s[i].end$
RECEIVE EVENT:
for $j := 1$ ***to*** N **do**
$s[j].end := \max(r[j].end, s[j].end)$
$s[j].beg := \max(r[j].beg, s[j].beg)$
end
$s[i].end := s[i].end + 1$
$s[i].beg := s[i].beg$

Listing 1: **stamp**

3.2.2 stamp()

For a given time stamp s on process p_i, the **stamp** algorithm is defined as follows. Initially, all entries of s are precise intervals equal to $\langle 0, 0 \rangle$ except for the i^{th} entry which is set to $\langle 1, 1 \rangle$. During a local/send event, the i^{th} entry is incremented. That is, if r is the old stamp on p_i, the new stamp s is defined by:

$$\textbf{precise}(s[i]) \wedge s[i].val = r[i].val + 1$$

$$(\forall\, j \,:\, 1 \leq j \leq N \wedge j \neq i \,:\, s[j] = r[j]\ )$$

Upon receiving a time tag t, the max of the *beg* and *end* points of each entry is taken and the i^{th} entry is incremented.

$$\textbf{precise}(s[i]) \wedge s[i].val = max(r[i].val, t[i].end) + 1$$

$$(\forall\, j \,:\, 1 \leq j \leq N \wedge j \neq i \,:\, s[j] = \langle \max(r[j].beg, t[j].beg), \max(r[j].end, t[j].end) \rangle\ )$$

Listing 1 gives the pseudo-code of **stamp**.

3.2.3 tag

The goal of the **tag** algorithm is to construct the smallest time tag possible while allowing the receiving event to maintain its bound on imprecision. Informally, the time tag is constructed by iteratively adding the greatest precise intervals until the error of the time tag is below the imprecision bound. The common imprecise interval of the time tag is formed by taking the max *end* value and the min *beg* value of the remaining intervals not in the time tag. The pseudo-code for **tag** is Listing 2. The function $\textbf{jth\_max}(j, s)$ returns the index of the j^{th} greatest precise interval.

```
Data: s is the time stamp of the send event on p_i , t is the outgoing tag
for j := 1 to N do
    t[j] := ⟨0, 0⟩
end
minbeg := ( Min j : 1 ≤ j ≤ N : s[j].beg )
j := 1
k := jth_max(i, r)
while (N − j + 1) ∗ (s[k].end − minbeg) > K do
    t[k] := s[k]
    j := j + 1
    k := jth_max(j, s)
end
for y := 1 to N do
    if t[y] = ⟨0, 0⟩ then
        t[y].end = s[k].end
        t[y].beg = minbeg
    end
end
```

Listing 2: **tag**

3.2.4 Example

Figure 3.2.4 depicts two examples of a process executing three events: a local event, a receive event, and a send event. In this example, the process is p_3 and the bound on imprecision is 30. Figure 3.2.4(a) is an example where the receipt of a time tag increases the imprecision of a time stamp. Figure 3.2.4(b), on the other hand, is an example where the receipt of a time tag decreases the imprecision of a time stamp.

We discuss, in detail, Figure 3.2.4(b). First, observe the local time stamp on p_3. It maintains the previously mentioned invariants of time stamps; namely, the precise intervals are greater than the imprecise intervals, the imprecise intervals share the same *end* value, and the interval corresponding to p_3 is precise. Also note that the sum of the widths of the intervals is equal to 6 which is less than the imprecision bound. The arriving time tag also satisfies its corresponding invariants.

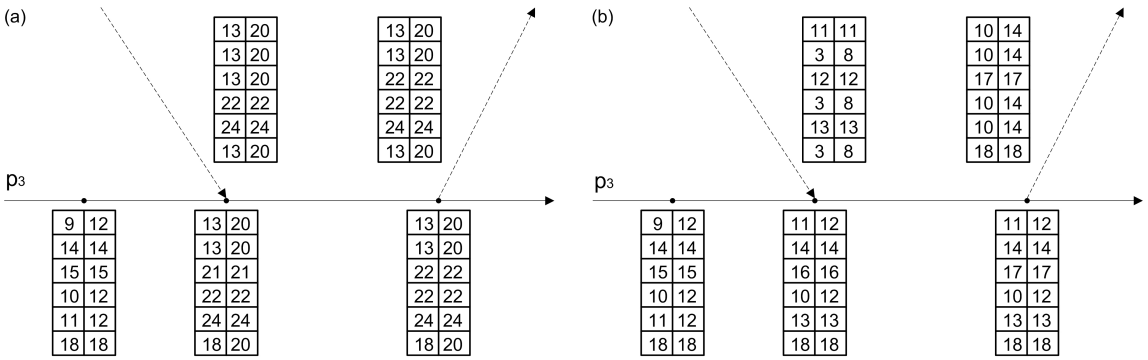


Figure 3.5: Example of Algorithm

Each imprecise interval is equivalent, and the precise intervals are greater than the imprecise intervals. The time stamp of the receive event is formed by taking the max *beg* and *end* values of the local time stamp and the incoming time tag. Finally, the interval corresponding to p_3 is incremented. Observe that the imprecision has decreased from 6 to 3.

The next event is the send event. First, the time stamp is created by incrementing the interval corresponding to p_3. The time tag is constructed as follows. First, we compute the minimal *beg* value of the time stamp. In this case, it is 10. We begin by considering the largest precise interval, $\langle 18, 18 \rangle$. We determine the resulting error of the time tag if we used $\langle 10, 18 \rangle$ as the common imprecise interval. In this case, it would be $6 * (18 - 10) = 48$. Since $48 > 30$, we add $\langle 18, 18 \rangle$ to the time tag and consider the next largest precise interval. The next largest interval is $\langle 17, 17 \rangle$ and the resulting error would be $5 * (17 - 10) = 35 > 30$. We again add the interval to the time tag and consider the next. At this point, we consider the interval $\langle 14, 14 \rangle$.

The resulting error is $4 * (14 - 10) = 16$ which is less than our imprecision bound. Therefore, we set the remaining intervals to $\langle 10, 14 \rangle$.

CHAPTER 4

Proofs of Correctness

There are two proof requirements for the correctness of P. First, we must show that P is plausible. Second, we must show that the imprecision of each time stamp is less than or equal to the given bound.

This chapter is outlined as follows. We begin by proving several invariants of time stamps and time tags. We show that our algorithm produces well-formed time stamps and time tags, and that these maintain a bound on error. We prove that our algorithm is plausible and that our time stamps maintain a bound on imprecision.

4.1 Recharacterization of S and G

In order to prove that the time stamps and time tags of our algorithm are well-formed, we first recharacterize S and G. We define **min_beg**(s) and **min_end**(s) as the minimum *beg* value and minimum *end* value of a time stamp s, respectively.

Definition 1.

$$\begin{aligned} \textbf{min\_beg}(s) &= (\,\textbf{Min}\, i \,:\, 1 \le i \le N \,:\, s[i].beg\,) \\ \textbf{min\_end}(s) &= (\,\textbf{Min}\, i \,:\, 1 \le i \le N \,:\, s[i].end\,) \end{aligned}$$

We can use **min_beg** and **min_end** to redefine S and G. S is defined as those elements s whose imprecise intervals have an *end* value equal to **min_end**(s):

$$(\forall i : 1 \leq i \leq N : \neg\textbf{precise}(s[i]) \Rightarrow s[i].end = \textbf{min\_end}(s)) \quad (4.1)$$

While G is defined as those elements t whose imprecise intervals share the same *end* value (4.1) and have a *beg* value equal to **min_beg**:

$$(\forall i : 1 \leq i \leq N : \neg\textbf{precise}(s[i]) \Rightarrow s[i].beg = \textbf{min\_beg}(s)) \quad (4.2)$$

With this recharacterization, we can prove several properties about time stamps. For instance, if a time stamp s has an imprecise interval, then **min_beg**(s) is strictly less than **min_end**(s).

$$(\exists i : 1 \leq i \leq N : \neg\textbf{precise}(s[i])) \Rightarrow \textbf{min\_beg}(s) < \textbf{min\_end}(s) \quad (4.3)$$

Proof. Let $s[i]$ be an time interval of s such that $\neg$**precise**$(s[i])$.

min_beg(s)

$=$ { Definition of **min_beg**(s) }

$(\textbf{Min}\, j : 1 \leq j \leq N : s[j].beg)$

$\leq$ { One-point rule }

$s[i].beg$

$<$ { Definition of $\neg$**precise**$(s[i])$ }

$s[i].end$

$=$ { *end* of imprecise interval is **min_end** }

min_end(s)

□

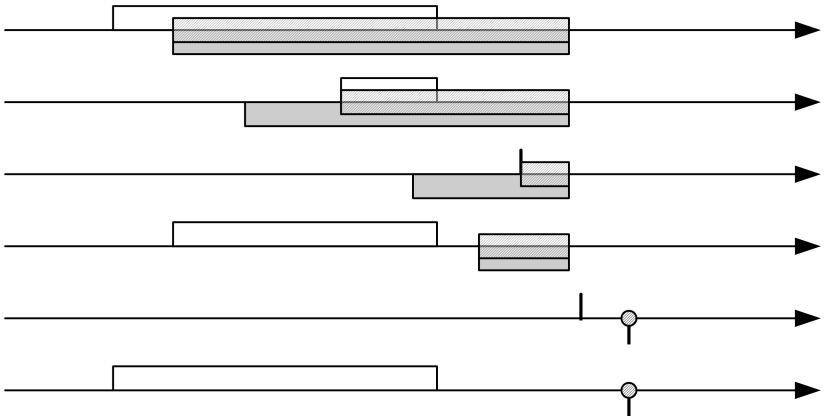


Figure 4.1: The join ($*$) operator

4.2 The join ($*$) operator

We define the *join* ($r * s$) of two time stamps by,

$$(\forall\, i \,:\, 1 \le i \le N \,:\quad (r*s)[i].beg = \mathbf{max}(r[i].beg, s[i].beg) \wedge$$
$$(r*s)[i].end = \mathbf{max}(r[i].end, s[i].end)\ )$$

Observe that we can redefine the **stamp** procedure in terms of $*$. For a receive event, the join of the old time stamp and the incoming time tag t is taken ($r * t$), and the local entry is incremented. Figure 4.2 is a graphical representation of the join operator. The white and gray imprecise intervals correspond to the two time stamps being joined. The dashed intervals (and circles to represent precise intervals) are the join of the two time stamps.

We can prove the following useful properties of join. First, the join operator is monotonic. That is, the join of two time stamps is greater than or equal to those

time stamps in terms of its value of **min_beg** and **min_end**.

$$\mathbf{min\_beg}(r * s) \geq \mathbf{max}(\mathbf{min\_beg}(r), \mathbf{min\_beg}(s)) \tag{4.4}$$

$$\mathbf{min\_end}(r * s) \geq \mathbf{max}(\mathbf{min\_end}(r), \mathbf{min\_end}(s)) \tag{4.5}$$

Proof. $\mathbf{min\_beg}(r * s) \geq \mathbf{max}(\mathbf{min\_beg}(r), \mathbf{min\_beg}(s))$

$$\begin{array}{ll} & \mathbf{min\_beg}(r * s) \\ = & \quad \{ \text{ Definition of } \mathbf{min\_beg} \ \} \\ & (\,\mathbf{Min}\, i \ :\ 1 \leq i \leq N \ :\ (r * s)[i].beg\ ) \\ \geq & \quad \{ \text{ Definition of } * \ \} \\ & (\,\mathbf{Min}\, i \ :\ 1 \leq i \leq N \ :\ r[i].beg\ ) \\ = & \quad \{ \text{ Definition of } \mathbf{min\_beg} \ \} \\ & \mathbf{min\_beg}(r) \end{array}$$

□

Proof. $\mathbf{min\_end}(r * s) \geq \mathbf{max}(\mathbf{min\_end}(r), \mathbf{min\_end}(s))$

Similar to above. □

Also, an interval that is precise in both time stamps is precise in the join of those time stamps.

$$(\forall\, i \ :\ 1 \leq i \leq N \ :\ \mathbf{precise}(r[i]) \wedge \mathbf{precise}(s[i]) \Rightarrow \mathbf{precise}((r * s)[i])\ ) \tag{4.6}$$

Proof.

$$\begin{array}{ll} & \mathbf{precise}(r[i]) \ \wedge\ \mathbf{precise}(s[i]) \\ \Rightarrow & \quad \{ \text{ Definition of } \mathbf{precise} \ \} \\ & r[i].beg = r[i].end \ \wedge\ s[i].beg = s[i].end \end{array}$$

$\Rightarrow$ { Leibniz }

$\mathbf{max}\,(r[i].beg, s[i].beg) = \mathbf{max}\,(r[i].end, s[i].end)$

$\equiv$ { Definition of $*$ }

$(r * s)[i].beg = (r * s)[i].end$

$\equiv$ { Definition of **precise** }

$\mathbf{precise}(r * s)[i]$

□

The imprecise intervals of the join of two time stamps share the same *end* value, and that *end* value is equal to the max of the **min_end** of both time stamps.

$$(\forall\, i \,:\, 1 \leq i \leq N \,:\, \neg\mathbf{precise}((r * s)[i]) \Rightarrow (r * s)[i].end = \mathbf{max}(\mathbf{min\_end}(r), \mathbf{min\_end}(s)\ ) \tag{4.7}$$

Proof. We will prove by contradiction. Assume there exists an i such that $\neg\mathbf{precise}(r * s)[i]$.

$(r * s)[i].end \neq \mathbf{max}\,(\mathbf{min\_end}(r), \mathbf{min\_end}(s))$

$\equiv$ { Monotonicity of $*$, (4.5) }

$(r * s)[i].end > \mathbf{max}\,(\mathbf{min\_end}(r), \mathbf{min\_end}(s))$

$\equiv$ { Property of **max** }

$(r * s)[i].end > \mathbf{min\_end}(r) \ \wedge\ (r * s)[i].end > \mathbf{min\_end}(s)$

$\equiv$ { Definition of $*$ }

$\mathbf{max}\,(r[i].end, s[i].end) > \mathbf{min\_end}(r)$

$\wedge\ \mathbf{max}\ (r[i].end, s[i].end) > \mathbf{min\_end}(s)$

$\Rightarrow$ { Property of **max** }

$r[i].end > \mathbf{min\_end}(r) \ \wedge\ s[i].end > \mathbf{min\_end}(s)$

$\Rightarrow$ { *end* of imprecise interval is **min_end** (4.1) }

precise$(r[i])$ $\wedge$ **precise**$(s[i])$

$\Rightarrow$ { precise interval of both time stamps is precise in join (4.6) }

precise$((r * s)[i])$

□

An important property is that S is closed under join. That is, the join of two time stamps is also a time stamp:

$$(\forall\, r, s \;::\; (r * s)[i].beg \leq (r * s)[i].end \wedge$$
$$\neg\textbf{precise}((r * s)[i]) \Rightarrow (r * s)[i].end = \textbf{min\_end}(r * s)\ )$$

Proof. We will prove each conjunct separately.

1. $(r * s)[i].beg \leq (r * s)[i].end$

$(r * s)[i].beg$

$=$ { Definition of $*$ }

$\textbf{max}\,(r[i].beg, s[i].beg)$

$\leq$ { Definition of a time interval }

$\textbf{max}\,(r[i].end, s[i].end)$

$=$ { Definition of $*$ }

$(r * s)[i].end$

2. $\neg\textbf{precise}((r * s)[i]) \Rightarrow (r * s)[i].end = \textbf{min\_end}(r * s)$

$\neg\textbf{precise}((r * s)[i])$

$\Rightarrow$ { Property of imprecise intervals of join (4.7) }

$$
\begin{array}{ll}
 & (r * s)[i].end = \ \mathbf{max}\,(\mathbf{min\_end}(r), \mathbf{min\_end}(s)) \\
\equiv & \quad \{\ \text{Definition of } \mathbf{min\_end}\ \} \\
 & (r * s)[i].end = \ \mathbf{max}\,(\,(\,\mathbf{Min}\, j\ :\ 1 \le j \le N\ :\ r[j].end\ ), \\
 & \qquad (\,\mathbf{Min}\, j\ :\ 1 \le j \le N\ :\ s[j].end\ )) \\
\Rightarrow & \quad \{\ \text{min max}\ \} \\
 & (r * s)[i].end \le\ (\,\mathbf{Min}\, j\ :\ 1 \le j \le N\ :\ \ \mathbf{max}\ (r[j].end, s[j].end)\ ) \\
\equiv & \quad \{\ \text{Definition of } *\ \} \\
 & (r * s)[i].end \le\ (\,\mathbf{Min}\, j\ :\ 1 \le j \le N\ :\ (r * s)[j].end)\ ) \\
\equiv & \quad \{\ \text{Property of } \mathbf{min}\ \} \\
 & (r * s)[i].end = \ (\,\mathbf{Min}\, j\ :\ 1 \le j \le N\ :\ (r * s)[j].end\ ) \\
\equiv & \quad \{\ \text{Definition of } \mathbf{min\_end}\ \} \\
 & (r * s)[i].end = \mathbf{min\_end}(r * s)
\end{array}
$$

□

4.3 Additional Properties of join

Using the closure property of join, we are able to prove several more properties. First, if there is an imprecise interval in the join of two time stamps, then the **min_end** of the join is equal to the max of the **min_end**'s of the time stamps.

$$
(\,\exists\, i\ ::\ \neg\mathbf{precise}((r * s)[i])\ ) \Rightarrow \\
\qquad \mathbf{min\_end}(r * s) = \mathbf{max}(\mathbf{min\_end}(r), \mathbf{min\_end}(s)) \qquad (4.8)
$$

Proof. $\neg\mathbf{precise}((r * s)[i])$

$$
\begin{array}{ll}
\equiv & \quad \{\ \text{Property of imprecise intervals of join (4.7)}\ \} \\
 & \neg\mathbf{precise}((r * s)[i])\ \wedge\ (r * s)[i].end = \ \mathbf{max}\,(\mathbf{min\_end}(r), \mathbf{min\_end}(s))
\end{array}
$$

$\Rightarrow$ { *end* of imprecise intervals is **min_end** (4.1) }

$(r * s)[i].end = \mathbf{min\_end}(r * s) \wedge (r * s)[i].end =$

$\mathbf{max}\,(\mathbf{min\_end}(r), \mathbf{min\_end}(s))$

$\Rightarrow$

$\mathbf{min\_end}(r * s) = \mathbf{max}\,(\mathbf{min\_end}(r), \mathbf{min\_end}(s))$

□

If an interval is imprecise in the join of two time stamps, it must have been imprecise in time stamp with the greatest value of **min_end**.

$$\neg\mathbf{precise}((r * s)[i]) \wedge \mathbf{min\_end}(r) \leq \mathbf{min\_end}(s) \Rightarrow \neg\mathbf{precise}(s[i]) \quad (4.9)$$

Proof. $\neg\mathbf{precise}((r * s)[i]) \wedge \mathbf{min\_end}(r) \leq \mathbf{min\_end}(s)$

$\Rightarrow$ { Property of **min_end** of join (4.8) }

$\neg\mathbf{precise}((r * s)[i]) \wedge \mathbf{min\_end}(r * s) = \mathbf{min\_end}(s)$

$\equiv$ { Imprecise interval implies **min_end** $<$ **min_beg** (4.3) }

$(r * s)[i].beg < \mathbf{min\_end}(r * s) \wedge \mathbf{min\_end}(r * s) = \mathbf{min\_end}(s)$

$\Rightarrow$

$(r * s)[i].beg < \mathbf{min\_end}(s)$

$\equiv$ { Definition of $*$ }

$\mathbf{max}\,(r[i].beg, s[i].beg) < \mathbf{min\_end}(s)$

$\Rightarrow$ { Property of **max** }

$s[i].beg < \mathbf{min\_end}(s)$

$\equiv$ { Definition of imprecise interval }

$\neg\mathbf{precise}(s[i])$

□

We will now prove that the time stamps and time tags of our algorithm are well-formed.

Theorem 1. For all time stamps s, the following holds:

$$(\forall\, i \,:\, 1 \leq i \leq N \,:\, \neg\textbf{precise}(s[i]) \Rightarrow s[i].end = \textbf{min\_end}(s)\ ) \quad (4.10)$$

For all $t \in G$, t satisfies (1) and (2), and the following:

$$(\forall\, i \,:\, 1 \leq i \leq N \,:\, \neg\textbf{precise}(t[i]) \Rightarrow t[i].beg = \textbf{min\_beg}(t)\ ) \quad (4.11)$$

Proof. We will prove by induction.

Base case - holds trivially. There are no time tags in the system. All time stamps have solely precise intervals, and their value is greater than or equal to $\textbf{min\_end}(s) = 0$.

Inductive step - assume the invariants hold for a history. Let us extend the history by a single event a on processor p_i. Let r be the stamp of the event immediately preceding a on p_i. There are three cases:

1. a is a local event

   There is no tag generated, so the tag invariants hold trivially. The time stamp invariants hold since the new stamp, s, is the same as r except for the i^{th} entry which is increased and maintained to be precise. Therefore, it will be precise, and its value will be larger than $\textbf{min\_end}(s)$.

2. a is a send event

   The time stamp invariants hold due to the previous argument (case 1). The new time tag, t, built from s is also a valid time tag. The *end* values of s

are taken in decreasing order, and given equal *beg* points. So, any imprecise interval has the **min_end** as its end point and uses **min_beg** as its *beg* point.

3. a is a receive event

   By the inductive assumption, both r and the incoming time tag, t, satisfy the invariants. We construct the new time stamp s by taking the join of r and t and incrementing its i^{th} entry. Since S is closed under join, and the i^{th} entry is increased and maintained as precise, s is a valid time stamp.

□

4.4 expand

The error of a time stamp is related to the sum of the widths of its time intervals. At the end of this chapter, we will show that this is in fact the imprecision of the time stamp. However, first we will show that this sum is bounded for all time stamps and time tags. We begin by defining the **expand** of a time stamp. Intuitively, the **expand** of a time stamp is equal to the imprecision of the worst-case time tag formed from that stamp. That is, the time tag that includes every precise interval of the time stamp. Figure 4.4 is a graphical representation of **expand**. The white imprecise intervals are from the original time stamp. The dashed imprecise intervals represent the corresponding widths used in **expand**.

Definition 1.

$$\mathbf{expand}(s) = (\sum i \ : \ 1 \leq i \leq N \wedge \neg \mathbf{precise}(s[i]) \ : \ \mathbf{min\_end}(s) - \mathbf{min\_beg}(s) \ )$$

We can show that the join operation does not increase the error of the system. That is, the **expand** of the join of two time stamps is less than or equal to the max

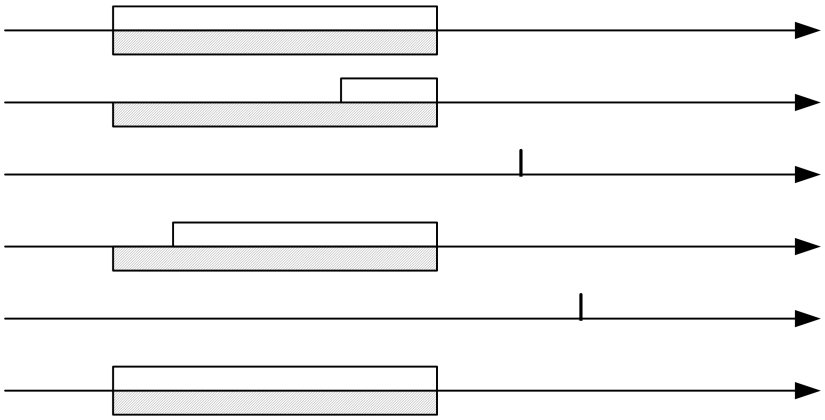


Figure 4.2: **expand**

of their respective **expand**'s.

$$\mathbf{expand}(r * s) \leq \mathbf{max}(\mathbf{expand}(r), \mathbf{expand}(s)) \tag{4.12}$$

Proof. $\mathbf{expand}(r * s) \leq \mathbf{max}\,(\mathbf{expand}(r), \mathbf{expand}(s))$

$\equiv$ { Definition of **max** }

$\mathbf{expand}(r * s) \leq \mathbf{expand}(r) \ \vee \ \mathbf{expand}(r * s) \leq \mathbf{expand}(s)$

Assume without loss of generality that $\mathbf{min\_end}(r) \leq \mathbf{min\_end}(s)$. It suffices to show that, under this assumption, $\mathbf{expand}(r * s) \leq \mathbf{expand}(s)$.

$\mathbf{expand}(r * s)$

$=$ { Definition of expand }

$(\sum i \ : \ 1 \leq i \leq N \ \wedge \ \neg\mathbf{precise}((r * s)[i]) \ : \ \mathbf{min\_end}(r * s) -$

$\mathbf{min\_beg}(r * s)\ )$

$=$ { w.l.o.g. }

$(\sum i \ : \ 1 \leq i \leq N \ \wedge \ \neg\mathbf{precise}((r * s)[i]) \wedge$

$$\mathbf{min\_end}(r) \leq \mathbf{min\_end}(s) \ : \ \mathbf{min\_end}(r * s) - \mathbf{min\_beg}(r * s) \ )$$

$=$ { Property of **min_end** of join (4.8) }

$$(\sum i \ : \ 1 \leq i \leq N \ \wedge \ \neg\mathbf{precise}((r * s)[i]) \wedge$$
$$\mathbf{min\_end}(r) \leq \mathbf{min\_end}(s) \ : \ \mathbf{min\_end}(s) - \mathbf{min\_beg}(r * s) \ )$$

$\leq$ { Imprecise interval in join is imprecise in time stamp (4.9) }

$$(\sum i \ : \ 1 \leq i \leq N \ \wedge \ \neg\mathbf{precise}(s[i]) \ : \ \mathbf{min\_end}(s) - \mathbf{min\_beg}(r * s) \ )$$

$\leq$ { Monotonicity of join (4.4) }

$$(\sum i \ : \ 1 \leq i \leq N \ \wedge \ \neg\mathbf{precise}(s[i]) \ : \ \mathbf{min\_end}(s) - \mathbf{min\_beg}(s) \ )$$

$=$ { Definition of expand }

$$\mathbf{expand}(s)$$

□

We now will show that the **expand** of all time stamps and time tags is less than or equal to the specified bound, K.

Theorem 1.

$$\mathbf{expand}(s) \leq K$$

$$\mathbf{expand}(t) \leq K$$

Proof. We will prove by induction.

Base case - Holds trivially. There are no time tags, and the **expand** of all initial time stamps is 0 since all intervals are precise.

Inductive step - Assume the invariants hold for a history. Let us extend the history by a single event, a, on processor p_i. Let r be the stamp of the event immediately preceding a on p_i. There are three cases:

1. a is a local event

   There is no tag generated, so the tag invariants hold trivially. The time stamp invariants hold since the new stamp, s, is the same as r except for the i^{th} entry which is increased and kept as precise. Therefore, the i^{th} entry is still larger than **min_end**.

2. a is a send event

   The time stamp invariants hold due to the previous argument (case 1). Consider the construction of the time tag t from the time stamp s. Since s satisfies the invariants, in the worst case every precise interval of s will be included in t. That means, in the worst case, $\mathbf{expand}(t) = \mathbf{expand}(s)$. Therefore, the invariants hold for t.

3. a is a receive event

   There is no tag generated, so the tag invariants hold trivially. The new time stamp s, is constructed by taking the join of r and the incoming time tag, t, and increasing the i^{th} entry. By our inductive assumption, the **expand** of r and t are both less than or equal to K. By **??**, $\mathbf{expand}(s)$ is also less than or equal to K.

□

4.5 Proof of Plausibility

In order to prove that P is plausible, we begin by showing that the $\xrightarrow{P}$ relation holds between local events and send-receive pairs.

Theorem 2. If a and b both occur on a process p_i, $a \rightarrow b \Leftrightarrow a \xrightarrow{P} b$.

Proof. It suffices to show that for two consecutive events a and b, $(\forall\, j \,:\, 1 \leq j \leq N \,:\, \mathbf{stamp}(a)[j].beg \leq \mathbf{stamp}(b)[j].beg\ ) \wedge \mathbf{stamp}(a)[i].end < \mathbf{stamp}(b)[i].beg$. That is, it suffices to show that $a \rightarrow Pb$ holds. Observe that for a local or send event, all entries of $\mathbf{stamp}(b)$ are equal to those of $\mathbf{stamp}(a)$, except for the i^{th} which is incremented. For a receive event, the entries of $\mathbf{stamp}(b)$ are guaranteed to be greater than or equal to those of $\mathbf{stamp}(a)$ since we max the beg values. Therefore, the theorem holds. □

Theorem 3. If a is a send event on p_i and b is the corresponding receive event on p_j, $a \xrightarrow{P} b$.

Proof. Let $r = \mathbf{stamp}(a)$, $s = \mathbf{stamp}(b)$, and let t be the time tag sent from a to b. We can derive the following:

$$(\forall\, k \,:\, 1 \leq k \leq N \,:\, r[k].end \leq t[k].end\ ) \quad \text{, By construction of } t \tag{4.13}$$

$$(\forall\, k \,:\, 1 \leq k \leq N \,:\, t[k].end \leq s[k].end\ ) \quad \text{, Maxing of } end \text{ values} \tag{4.14}$$

$$(\forall\, k \,:\, 1 \leq k \leq N \,:\, r[k] \overset{interval}{\lessapprox} s[k]\ ) \quad \text{, By 4.13 and 4.14} \tag{4.15}$$

$$(\exists\, k \,:\, 1 \leq k \leq N \,:\, t[k].end < s[k].beg\ ) \quad \text{, Let } k = j \tag{4.16}$$

$$(\exists\, k \,:\, 1 \leq k \leq N \,:\, r[k] \overset{interval}{<} s[k]\ ) \quad \text{, By 4.13 and 4.16} \tag{4.17}$$

$$a \xrightarrow{P} b \quad \text{, By 4.15 and 4.17} \tag{4.18}$$

□

Another property of P is that precise intervals imply a causal relationship. That is, if a time stamp has a precise interval it must have been propagated from the original event.

Theorem 4. $(\forall\, a \in H_i, b \in H_j \;::\; a \neq b \;\wedge\; \mathbf{stamp}(a)[i] \overset{interval}{=} \mathbf{stamp}(b)[i] \Rightarrow a \rightarrow b\,)$.

Proof. Let $r = \mathbf{stamp}(a)$ and $s = \mathbf{stamp}(b)$. We will prove by contradiction. Assume $a \neq b \;\wedge\; r[i] \overset{interval}{=} s[i] \;\wedge\; \neg(a \rightarrow b)$. It suffices show that it is impossible for $s[i] = r[i]$. Consider how the value of $s[i]$ was obtained. The value was either created at s, or propagated to s. Let's consider the point of creation. That is, there exists a first time stamp $q = \mathbf{stamp}(c)$ such that $c \rightarrow b$ and $q[i] = r[i]$. Formally, $(\exists\, c \;::\; c \rightarrow b \;\vee\; c = b \;\wedge\; q[i] = r[i] \;\wedge\; \neg\,(\exists\, d \;::\; d \rightarrow c \;\wedge\; \mathbf{stamp}(d)[i] = r[i]\;)\;)$. It suffices to show that is impossible for $q[i] = r[i] \;\wedge\; a \neq c$. There are three cases:

1. c is a local or send event

   The **stamp** function only modifies the interval corresponding to a stamp's process. If the value of $q[i]$ was propagated from an event local to c, it would violate the assumption that q is the first such time stamp with the value $q[i] = r[i]$. Also, c could not have occurred on process p_i and still maintain $a \neq c$ because the interval corresponding to a time stamp's process is strictly increasing.

2. c is a receive event, and the value of $q[i]$ was propagated through the incoming time tag

   An interval of a time tag is only precise if it was precise in the corresponding send event. This would violate our assumption that q is the first such time stamp with the value $q[i] = r[i]$.

3. c is a receive event, and the value of $q[i]$ occurred during the merging of data

The value of $q[i]$ is precise. Observe that the only way for an interval to be precise after a receive event is if it was precise in either the previous local time stamp or the incoming time tag. Therefore, this is impossible.

In all cases, there is a contradiction. Therefore, the lemma holds. □

Definition 2. $\mathbf{latest}(H_i) = a : (\forall b \in H_i :: b \rightarrow a \vee b = a)$

Lemma 1. $(\forall H_i, a : a = \mathbf{latest}(H_i) : (\forall b :: \mathbf{stamp}(b)[i].beg \leq \mathbf{stamp}(a)[i].beg))$

Proof. We will prove by induction. Initially, $(\forall a \in H_i :: \mathbf{stamp}(a)[i].beg = 1 \wedge (\forall j : 1 \leq j \leq N : \mathbf{stamp}(a)[j].beg \leq 1))$, therefore the invariant holds. Assume for all events, the invariant holds. Let a on process p_i be the next event in the system. There are two cases:

1. a is a send or local event

   Let c be the event immediate preceding a on p_i. By definition of the **stamp** function, we know $\mathbf{stamp}(a)[i].beg \geq \mathbf{stamp}(c)[i].beg \wedge (\forall j : 1 \leq j \leq N : \mathbf{stamp}(a)[i] = \mathbf{stamp}(c)[i])$. Since we assume that c maintains the invariant, the following holds:

   $$(\forall b :: \mathbf{stamp}(b)[i].beg \leq \mathbf{stamp}(c)[i].beg \leq \mathbf{stamp}(a)[i].beg) \wedge$$
   $$(\forall H_j, b : b = \mathbf{latest}(H_j) : \mathbf{stamp}(a)[j].beg = \mathbf{stamp}(c)[j].beg \leq \mathbf{stamp}(b)[i].beg)$$

   Therefore, the invariant holds.

2. a is a receive event

Let c be the event immediately preceding a on p_i. Let t be the message a receives from event d. By definition of the **tag** function, we know $(\forall\, j\, :\, 1 \leq j \leq N\, :\, t[j].beg \leq \mathbf{stamp}(d)[j].beg\,)$. Therefore, by the definition of **stamp**, we know the following:

$$(\forall\, j\, :\, 1 \leq j \leq N \wedge j \neq i\, :\, \mathbf{stamp}(a)[j].beg \leq \max(\mathbf{stamp}(c)[j].beg, \mathbf{stamp}(d)[i].beg)\,) \wedge \mathbf{stamp}(a)[i].beg \geq \mathbf{stamp}(c)[i].beg$$

By our assumption that c and d maintain the invariants, the following holds:

$$(\forall\, b\, ::\, \mathbf{stamp}(b)[i].beg \leq \mathbf{stamp}(c)[i].beg \leq \mathbf{stamp}(a)[i].beg\,) \wedge$$
$$(\forall\, H_j, b\, :\, b = \mathbf{latest}(H_j, b)\, :\, \mathbf{stamp}(a)[j].beg \leq \max(\mathbf{stamp}(c)[j].beg, \mathbf{stamp}(d)[j].beg) \leq \mathbf{stamp}(b)[i].beg\,)$$

□

Similarly to the presence of precise intervals, an interval of one time stamp being strictly less than the corresponding interval of another time stamp also implies a causal relationship.

Theorem 5. Let a and b be events on processes p_i and p_j respectively. The following holds: $\mathbf{stamp}(a)[i] \overset{interval}{<} \mathbf{stamp}(b)[i] \Rightarrow a \rightarrow b$.

Proof. We will prove by induction. Initially, there does not exist two events a on p_i and b on p_j such that $\mathbf{stamp}(a)[i] \overset{interval}{<} \mathbf{stamp}(b)[i]$. Therefore, the invariant trivially holds. Assume that for all events, the invariant holds. Let a on p_i be the next event in the system. There are two cases:

1. a is a send or local event

   Let c be the event immediately preceding a on p_i. By the **stamp** function, we know $(\forall b \in H_j : j \neq i : (\mathbf{stamp}(b)[j] \overset{interval}{<} \mathbf{stamp}(c)[j] = \mathbf{stamp}(a)[j]) \Rightarrow b \rightarrow c \rightarrow a)$. Since **stamp** increments the i^{th} entry of $\mathbf{stamp}(c)$ and $c \rightarrow a$, we know $(\forall b \in H_i :: \mathbf{stamp}(b)[i] \overset{interval}{<} \mathbf{stamp}(a)[i] \Rightarrow b \rightarrow a)$. What remains to be shown is $(\forall b :: \mathbf{stamp}(a)[i] \overset{interval}{<} \mathbf{stamp}(b)[i] \Rightarrow a \rightarrow b)$. However, by Lemma 1 we know $\neg(\exists b :: \mathbf{stamp}(a)[i] \overset{interval}{<} \mathbf{stamp}(b)[i])$, and therefore it is trivially true.

2. a is a receive event

   Let c be the event immediately preceding a on p_i. Let t be the message a receives from event d. By the **tag** function, we know $(\forall k : 1 \leq k \leq N : t[k].beg \leq \mathbf{stamp}(d)[k].beg)$. By the **stamp** function, we know $(\forall k : 1 \leq k \leq N : \mathbf{stamp}(a)[k].beg \leq \max(\mathbf{stamp}(c)[k].beg, \mathbf{stamp}(d)[k].beg)$. $(\forall b \in H_j : j \neq i : \mathbf{stamp}(b)[j] \overset{interval}{<} \mathbf{stamp}(a)[j] \Rightarrow (\mathbf{stamp}(b)[j] \overset{interval}{<} \mathbf{stamp}(c)[j] \vee \mathbf{stamp}(b)[j] \overset{interval}{<} \mathbf{stamp}(d)[j]))$. Since **stamp** increments the i^{th} entry of $\mathbf{stamp}(c)$ and $c \rightarrow a$, we know $(\forall b \in H_i :: \mathbf{stamp}(b)[i] \overset{interval}{<} \mathbf{stamp}(a)[i] \Rightarrow b \rightarrow a)$. What remains to be shown is $(\forall b :: \mathbf{stamp}(a)[i] \overset{interval}{<} \mathbf{stamp}(b)[i] \Rightarrow a \rightarrow b)$. However, by Lemma 1 we know $\neg(\exists b :: \mathbf{stamp}(a)[i] \overset{interval}{<} \mathbf{stamp}(b)[i])$, and therefore it is trivially true.

□

Theorem 6. P is plausible.

Proof. There are two proof obligations:

1. $a \rightarrow b \Rightarrow a \xrightarrow{P} b$

   Assume $a \rightarrow b$; therefore, there exists a chain of events $c_0, c_1, ..., c_n$ where $c_0 = a$ and $c_n = b$ and $(\forall i : 0 \leq i < n : c_i \rightarrow c_{i+1})$. It suffices to show the following properties:

   (a) $(\forall i : 0 \leq i < n : c_i \xrightarrow{P} c_{i+1})$

   There are two cases: c_i and c_{i+1} are either local to each other or a send receive pair. Both of these cases are directly proved by Theorem 2 and 3.

   (b) $(\forall i, j, k : 0 \leq i < j < k \leq n : (c_i \xrightarrow{P} c_j \wedge c_j \xrightarrow{P} c_k) \Rightarrow c_i \xrightarrow{P} c_k)$

   Let $s_i = \textbf{stamp}(c_i)$, $s_j = \textbf{stamp}(c_j)$, and $s_k = \textbf{stamp}(c_k)$. Observe that the *end* points of intervals are non-decreasing in our system (we either increment or max them), therefore, we can derive the following:

$$(\forall m : 1 \leq m \leq N : s_i[m].end \leq s_j[m].end \leq s_k[m].end)$$

$$\text{, } end \text{ points are non-decreasing} \tag{4.19}$$

$$\neg(\exists m : 1 \leq m \leq N : s_k[m] \overset{interval}{<} s_i[m])$$

$$\text{, By 4.19} \tag{4.20}$$

$$(\exists m : 1 \leq m \leq N : s_j[m].end < s_k[m].beg)$$

$$\text{, By definition of } \xrightarrow{P} \text{ and } \rightarrow \tag{4.21}$$

Let m be an integer such that $s_j[m].end < s_k[m].beg$ (by 4.21). We can derive the following:

$$(\forall g \,:\, 1 \le g \le N \,:\, s_i[g] \overset{interval}{\lessapprox} s_k[g]\,)$$

, By 4.20 (4.22)

$$s_i[m].end \le s_j[m].end < s_k[m].beg$$

, By 4.19 and definition of m (4.23)

$$s_i[m] \overset{interval}{<} s_k s[m]$$

, By definition of $\overset{interval}{<}$ (4.24)

$$c_i \xrightarrow{P} c_k$$

, By 4.22 and 4.24 (4.25)

2. $a = b \iff a \overset{P}{=} b$

Observe that by our definition of the $\overset{interval}{\approx}$ relation and $\overset{P}{=}$, $a = b \Rightarrow a \overset{P}{=} b$ holds. We will prove by contradiction that $a \overset{P}{=} b \Rightarrow a = b$. Let p_i and p_j be the processes for a and b respectively. Assume $a \overset{P}{=} b \,\wedge\, a \neq b$. In order for $a \overset{P}{=} b$ to hold, the following must be true, $\mathbf{stamp}(a)[i] \overset{interval}{=} \mathbf{stamp}(b)[i] \,\wedge\, \mathbf{stamp}(b)[j] \overset{interval}{=} \mathbf{stamp}(a)[j]$. By applying Theorem 4, $a \rightarrow b$. However, we previously proved that $a \rightarrow b \Rightarrow a \xrightarrow{P} b$ which contradicts our assumption $a \overset{P}{=} b$.

□

4.6 Proof of Bounded Imprecision

Theorem 7.

$\psi(\mathbf{stamp}(a)) \le (\sum i \,:\, 1 \le i \le N \,:\, \mathbf{stamp}(a)[i].end - \mathbf{stamp}(a)[i].beg\,)$

Proof. We will prove by contradiction. Assume $\psi(\mathbf{stamp}(a)) > (\sum i : 1 \leq i \leq N : \mathbf{stamp}(a)[i].end - \mathbf{stamp}(a)[i].beg)$. By Theorem 4 and 5, $(\forall b \in H_j :: (\mathbf{stamp}(b)[j] \overset{interval}{<} \mathbf{stamp}(a)[j] \vee \mathbf{stamp}(b)[j] \overset{interval}{=} \mathbf{stamp}(a)[j]) \Rightarrow b \rightarrow a)$. Since events satisfying the $\overset{interval}{=}$ and $\overset{interval}{<}$ relations are correctly ordered, ψ must be a function of the events that are related by $\overset{interval}{\approx}$. Observe that the precise intervals corresponding to a process are strictly increasing on that process. Therefore, for a given process p_j, there can be at most $\mathbf{stamp}(a)[j].end - \mathbf{stamp}(a)[j].beg$ incorrectly related events ($\overset{P}{\rightarrow}$ but not $\rightarrow$). Therefore, $\psi(\mathbf{stamp}(a)) \leq (\sum i : 1 \leq i \leq N : \mathbf{stamp}(a)[i].end - \mathbf{stamp}(a)[i].beg)$ which is a contradiction. □

CHAPTER 5

Performance Evaluation

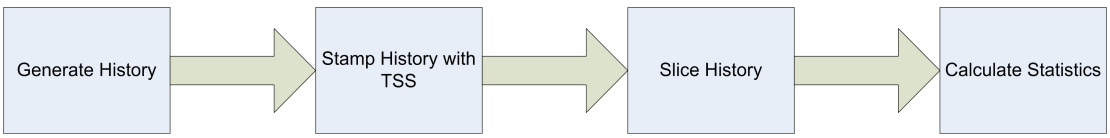
5.1 Simulation Framework

The experimental results presented in this chapter were obtained through a four step process. First, a history is generated according to several simulation parameters. Second, the plausible clock algorithms being evaluated are executed on the history. That is, for a given simulation, each plausible clock is executed on the *same* history. Third, a subset of the history is calculated. Finally, statistics are computed from the subset of the history. Figure 5.1 depicts this process.

5.1.1 Generating Histories

A history is generated by using a discrete event simulation written in C++. Events are placed in a single priority queue. When an event reaches the head of the queue,

Figure 5.1: Simulation Framework



```
INITIALLY:
    for j := 1 to N do
        event a := ⟨0.0, j, local⟩ ;
        p_queue.insert(a) ;
    end
begin
    event a := p_queue.dequeue() ;
    if a.type = SEND then
        event b ;
        b.type := receive ;
        b.pid := destination(a.pid) ;
        b.time := a.time + transmit_delay(a.pid, b.pid) ;
        p_queue.insert(b) ;
    end
    event b ;
    b.pid := a.pid ;
    b.type := event_rule(b.pid) ;
    b.time := a.time + local_delay(b.pid) ;
    p_queue.insert(b) ;
end
```

Listing 3: Generating Histories

it is evaluated and another event is scheduled. The simulation is parameterized by four procedures: *transmit_delay*, *local_delay*, *event_rule*, and *destination*. The procedure *transmit_delay* is used to model the delay of channels in the system. The *local_delay* procedure is used to model the varying speed of each process. When rescheduling another event for a given process, *event_rule* is used to dictate the type of the new event (send or local). Finally, *destination* is used to compute the target process of a send event.

The pseudo code for the history generation algorithm is presented in Listing 3. An event is a tuple $\langle time, pid, type \rangle$, where *time* is the physical time at which the event is executed, *pid* is the id of the event's process, and *type* is one of: *local*,

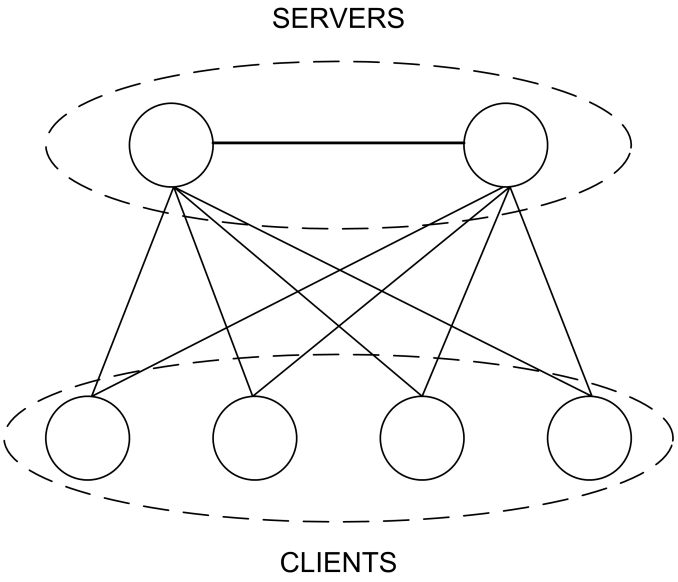


Figure 5.2: Client-Server System

send, *receive*. The priority queue *p_queue* is a queue of events. It maintains that the event whose *time* is the least is kept at the head of the queue.

5.1.2 Client-Server System

For our experiments, we consider a client-server system similar to the one used in previous plausible clock papers [16, 15]. There are two types of processes in the system: clients and servers. A client may communicate with any server of the system, but not other clients. A client sends a request to a server, and is not allowed to send another message until the server replies to the request. Servers may communicate with one another if they have no outstanding requests. A server replies to client requests in first-come first-serve order. Figure 5.1.1 depicts a client-server system. The system is defined by the following procedures:

- *transmit_delay*$(src, dest)$

  returns a constant delay of 1.0 .

- *local_delay*(src)

  returns a random number based upon a negative exponential distribution with a mean of 1.0 .

- *event_rule*(src)

  1. *src* is a client

     return a *local* event if waiting on an outstanding request. return a *send* event otherwise.

  2. *src* is a server

     return a *send* event.

- *destination*(src)

  1. *src* is a client

     return any server process id using a uniform random distribution.

  2. *src* is a server

     if there are no outstanding requests, return a server process id using a uniform random distribution. if there are outstanding requests, return the id of the next client to be serviced.

For the following experiments, we consider a 98-client, 2-server system.

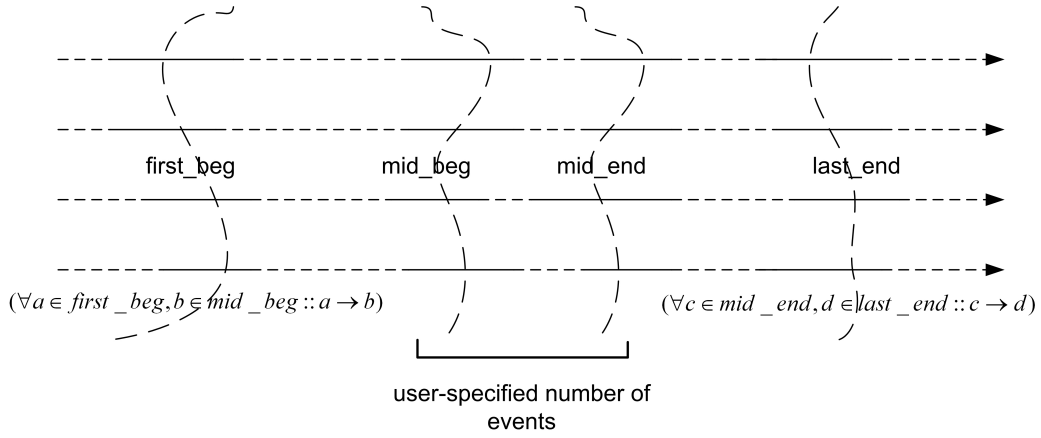


Figure 5.3: Cuts Required for a Middle Subset

5.1.3 Slicing Histories

Our goal is to construct an accurate expected-case analysis of our algorithm. In practice, time stamping schemes are executed on arbitrarily large histories. However, our generated histories are relatively small (less than 600 events per process). Furthermore, there is an initial startup delay in which processes begin to communicate with each other. In fact, our algorithm performs particularly well during this initial startup. Therefore, in order to eliminate this bias from our expected-case analysis, we consider a subset of events from the middle of the history.

The middle subset is constructed as follows. First, we construct the cut *start_beg* by selecting an event e from each process such that there is at least one event from each process that happens before e. In other words, we select the first event, from each process, whose vector clock time stamp would have no zero entries. We then construct the cut *mid_beg* by selecting the first event e from each process such that every event in *start_beg* happens before e. The cut *mid_end* is constructed by

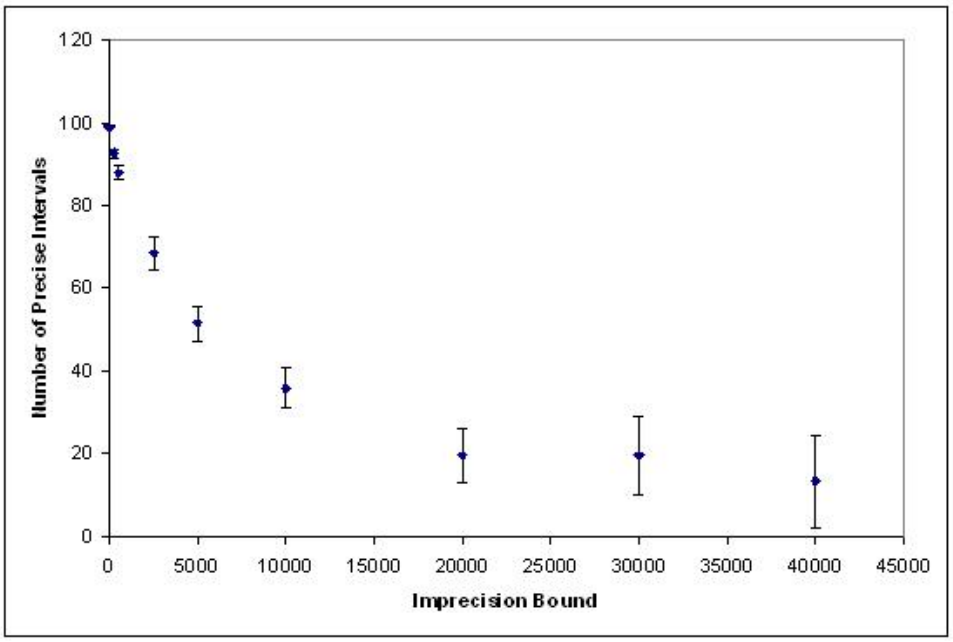


Figure 5.4: Imprecision and Message Size

taking the event from each process that is a user-specified number of events later than the process' event in *mid_beg*. Finally, the cut *last_end* is constructed by taking the first event e from each process such that all events in *mid_end* happen before e. Our middle subset contains the events between (and including) *start_beg* and *last_end*. Figure 5.1.2 depicts the process of constructing these cuts.

5.2 Experimental Results

5.2.1 Imprecision and Message Size

Our algorithm is parameterized by a bound on imprecision, and maintains that bound by allowing the size of time tags to grow and shrink throughout the course of

the computation. Figure 5.1.3 depicts the relationship between the specified imprecision bound and the resulting mean time tag size. Since this figure solely considers our algorithm, we define time tag size as the number of precise intervals included in the time tag.

Figure 5.1.3 shows that a greater bound on imprecision results in a decrease in time tag size. This isn't surprising. A greater imprecision bound allows for a common interval of greater length.

An interesting observation is that the drop in message size is not linear with respect to the increase of imprecision. A more formal analysis of this result is planned for future work, but consider the following hypothesis. We gain in time tag size by representing precise intervals with the common interval. As the imprecision bound increases, were are allowed to put precise intervals with greater values into the common interval. However, if those greatest precise intervals have a distribution that is more spread out than the rest, than the cost of putting those precise intervals in the common interval increases as our time tags get smaller (more precise intervals are put in the common interval). In this scenario, the rate of the decrease in message size would become smaller as the imprecision increases.

5.2.2 Inaccuracy Bound and Observed Inaccuracy

In this thesis, we've shown how imprecision can be used to define an upper bound on inaccuracy. An obvious question is what is the relationship between this upper bound and the actual inaccuracy. Figure 5.2.2 shows the relationship between the

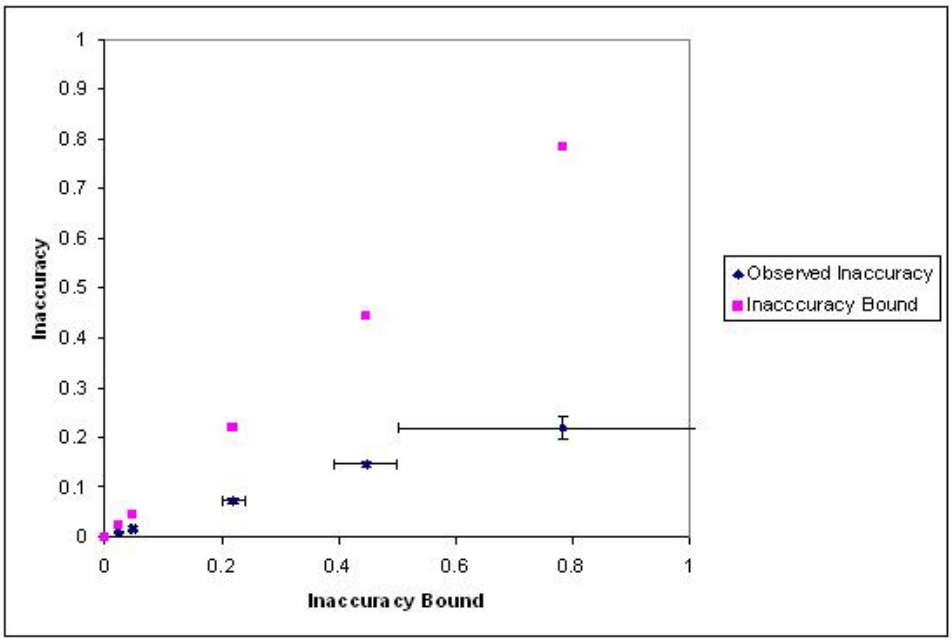


Figure 5.5: Inaccuracy Bound and Observed Inaccuracy

calculated inaccuracy bound and the actual observed inaccuracy. For these experiments, the value of the concurrency ratio $\epsilon(H)$ was computed individually for each run.

Observe that the calculated inaccuracy bound is not a tight bound. That is, there are cases where the actual inaccuracy is far lower than the bound. One reason for this is that imprecision reflects a worst-case error for a time stamp; a particular history may not be the worst-case for all time stamps. Also, even if our algorithm does not append any precise intervals to time tags (every time tag is just the common interval) it can still detect concurrency. Therefore, it will never result in an inaccuracy of 1.

5.2.3 Comparison with Existing Plausible Clocks

For the performance evaluation of our algorithm, we compared it with two previously proposed plausible clocks: REV and Comb [16]. Comb is a combination of two plausible clocks: REV and k-Lamport. For our experiments, we fixed the size of the k-Lamport part of Comb to 5. Therefore, for a given size, R, Comb is comprised of a k-Lamport clock of size 5 and a REV clock of size $R-5$. For our experiments, an integer of a time tag is encoded with 64 bits.

Figure 5.2.3 shows the results of our study. Observe that, in general, our algorithm performs better than both other clocks. This isn't surprising: our algorithm evaluates the information of its time stamps and then sends the smallest time tag possible. However, as the inaccuracy reaches zero, and the number of precise intervals required in our time tags becomes close to N, the cost of encoding the processor id with each precise interval causes our algorithm to be out-performed. But, we argue that if the application is willing to tolerate plausible clocks of that size, it might as well use vector clocks.

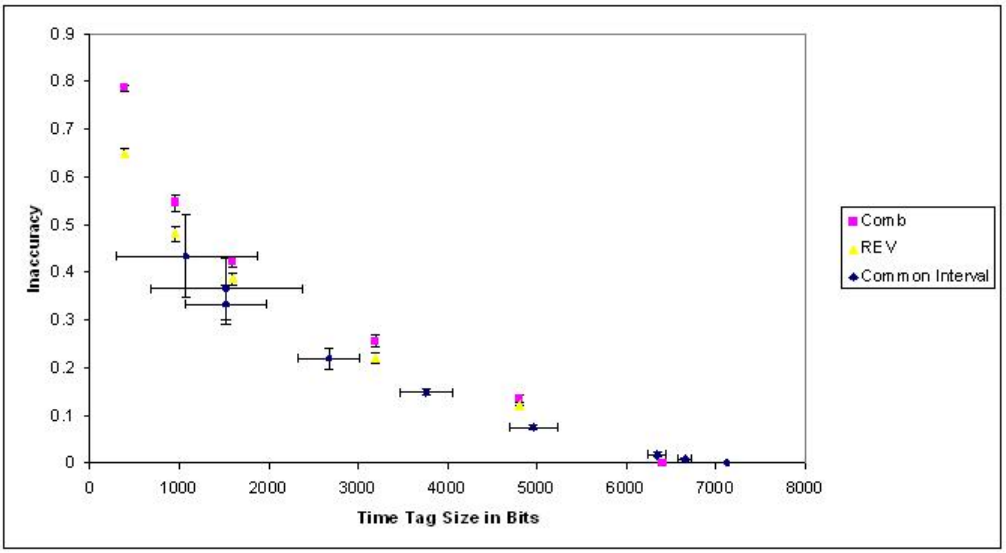


Figure 5.6: Performance Comparison

CHAPTER 6

Conclusion

6.1 Related Work

Several algorithms have been proposed as scalable solutions to vector clocks. For instance, in [2] Baldoni and Melideo proposed k-Dependency Vectors. Their algorithm affixed a constant-size vector of integers to application messages. The tradeoff of this approach was that, in certain cases, extra computation was required to detect the causal relationship between events. The algorithm required a dedicated checker process to determine the causal order of events. The above mentioned *detection delay* occurs when the checker process has to delay its decision until all relevant notifications have arrived.

Two plausible clock algorithms were presented: R-Entries Vector (REV) and k-Lamport. In [16, 15], the performance of the *Comb* (REV and k-Lamport combined) clock was analyzed through simulation. The results of those studies showed good performance of *Comb* and also the dependency of that performance on several factors (e.g., local history size, communication pattern, system size). However, formal analysis of the expected behavior of a plausible clock algorithm was left for future work.

In [8], Gidenstam and Papatriantafilou made several steps in the investigation of the issues proposed in [16]. They introduced NUREV clocks, a plausible clock algorithm with a constant number of vector entries and a dynamic mapping of processor-ids to those entries. They then formulated a formal quantification of the error produced by their algorithm. Using this analysis of error, they proposed several mapping algorithms to minimize the error produced by information loss. One such mapping was R-Others Vector-Most Recent Senders (ROV-MRS). Unlike our precision guaranteed clock, in which the R max time stamps are mapped to message entries, ROV-MRS maps the most recently heard from processors to R-1 time stamp entries. The motivation of this approach is to minimize the number of time stamps in which error is propagated.

Our approach to the worst-case analysis of plausible clocks differs from the contributions of [8] in several ways. First, the analysis of the run-time error of NUREV clocks was used to construct optimal mapping algorithms. *Imprecision*, on the other hand, is a means by which to evaluate the performance of plausible clocks independent of a specific history. Also, it provides a way in which an algorithm can (at run-time) evaluate the error it is producing and thereby control it.

6.2 Summary

Plausible clocks are used in situations where incorrectly ordering concurrent events impacts performance but does affect correctness. Previously proposed plausible clocks use a fixed message size and allow the number of incorrectly ordered concurrent pairs to vary from run to run. In this thesis, we have considered the problem of reasoning about and bounding the inaccuracy of plausible clocks. Specifically,

- We have defined the metric *imprecision*, which is a history-independent bound on the worst-case error of a time stamp,
- We have derived a bound on inaccuracy in terms of imprecision,
- We have introduced, and proven the correctness of, a plausible clock algorithm that is parameterized by an arbitrary bound on imprecision, and
- We have shown that our algorithm achieves good performance with respect to existing plausible clocks.

6.3 Future Work

The work presented in this thesis provides several possibilities for future work. For instance, in Chapter 5, we observed that for our algorithm, the observed inaccuracy may be far lower than the inaccuracy bound. The possibility exists that by storing more information with a time stamp regarding its imprecision, one may decrease this gap, thereby allowing the user of the clock to have a finer grain of control on error.

Another interesting idea would be to consider changing the bound on imprecision at run-time. For instance, in distributed debugging and visualization, a user may want a more detailed view of the system at critical points of the computation. Pursuit of this problem would require characterizing under what conditions the imprecision of the system would converge and/or modifying our algorithm to make convergence more likely.

BIBLIOGRAPHY

[1] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[2] R. Baldoni and G. Melideo. k-dependency vectors: A scalable causality-tracking protocol. In *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2003.

[3] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[4] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, 1991.

[5] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, Oct. 1996.

[6] A. Fernández, E. Jiménez, and V. Cholvi. On the interconnection of causal memory systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 163–170, July 2000.

[7] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11 Australian Computer Science Conference*, pages 55–66, 1988.

[8] A. Gidenstam and M. Papatriantafilou. Adaptive plausible clocks. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 86–93. IEEE Computer Society, 2004.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, 1985.

[11] F. Mattern. Virtual time and global states of distributed systems. In M. C. et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[12] R. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, 1995.

[13] R. Prakash, M. Raynal, and M. Singhal. An adaptive causal ordering algorithm suited to mobile computing environments. *Journal of Parallel and Distributed Computing*, 41(2):190–204, 1997.

[14] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, January 1981.

[15] F. J. Torres-Rojas. Performance evaluation of plausible clocks. In *Proceedings of the 7th Euro-Par Conference*, pages 476–481, 2001.

[16] F. J. Torres-Rojas and M. Ahamad. Plausible clocks: constant size logical clocks for distributed systems. *Distributed Computing*, 12(4):179–196, 1999.