Order Number 9501009

Synchronization of fault-tolerant distributed real-time multicomputers

Olson, Alan David, Ph.D.

The University of Michigan, 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

SYNCHRONIZATION OF FAULT-TOLERANT DISTRIBUTED REAL-TIME MULTICOMPUTERS

by

Alan David Olson

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1994

Doctoral Committee:
Professor Kang G. Shin, Chair
Professor John R. Birge
Assistant Professor Stuart Sechrest
Associate Research Scientist Chinya V. Ravishankar

To my parents

ACKNOWLEDGEMENTS

When I started I could not have known how much time and effort would go into the production of this dissertation. Many times along the way I thought either the end was near or the end would never come. Now that the end *is* here, I would like to take the time to thank some of the people who helped me along the way.

First of all I must thank my advisor, Professor Kang Shin, who put up with me and supported me these many years. I am amazed at the patience he showed, and the faith he had in me even when I seemed to be going nowhere. He could have yelled at me more often to keep me moving, but I have profited more by learning to yell at myself.

I must also give my heartfelt thanks to my committee, Professors Stuart Sechrest and John Birge, and Chinya Ravishankar, for their helpful comments and suggestions.

I am also deeply indebted to Martin Marietta corporation and Bruno Jambor, who provided much of the funds. They insisted I do work that I thought at the time to be useless, but would eventually form an important part of this dissertation.

I would never have been able to do much of the analysis in this dissertation were it not for the authors of both Mathematica and Maple V. While I often cursed these wonderful tools, I was always able to find a way.

I also owe much to the members of the Real-Time Computing Laboratory, past and present. Special thanks goes to James Dolter, who kept the systems running, wrote the LaTeX style I used for this dissertation, and answered the endless questions I had about TeX and everything else. Also worthy of special note is B.J. Monaghan, who filled out all the paperwork, kept the candy jar filled, and guided me through the maze of paperwork and bureaucracy that is the University.

Finally, I wish to thank Chuck, Don, Liz, Jon, and the rest of the people at DCO who were more than patient with me and my continual difficulties. If it weren't for them I would still be banging away on a typewriter somewhere.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Everyone who has ever driven a car has probably found himself waiting at a red light and looking ahead to see that all the lights up ahead are green, and then watching them turn red, one by one and in perfect sequence, usually just as the light he is waiting at changes back to green. Some may attribute this to sadistic tendencies on the part of traffic control engineers. Others might call it coincidence. A rare few might even believe there is a good reason for it. Almost nobody would attribute it to a loss of synchronization, but such may very well be the case.

The cycles on older traffic light systems, and modern systems whose control systems have failed, are controlled by simple timers. On a heavily-traveled street all the lights use the same cycle, and start their cycles in a staggered fashion, with the time between when two lights start their cycles being the approximate time needed to drive between them. The result is that traffic flows through unimpeded (at least in one direction). However, a light will slowly drift away from the nice staggered pattern if the timer that controls the cycle at the light runs either a little fast or slow. Such a light will eventually stop any cars that pass though the previous light, and let them through only when they will be stopped at the next light.

The problems of traffic control may seem remote from those of real-time systems, but there are parallels. Consider a situation where one node of the system, N_a, performs some computation and sends the result to another node, N_b, for further processing. Suppose N_a is late. When N_b receives the result, it may have already moved on to other tasks, and may not have the resources available to do the necessary processing. Processing the result sent by N_a may have to wait until the necessary resources become available, or until N_b can rearrange its schedule. If the result of the processing at N_b is sent somewhere else, the delay at N_b will make it even later to its next destination. N_a is analogous to the out-of-sequence

traffic light. Tasks arriving from other nodes are early from N_a's perspective, and are forced to wait. When the results are finally forwarded to the next node, they arrive late and are forced to wait again.

One might be tempted to say that the problems in the preceding example could have been avoided if N_b had simply informed N_a that it was expecting the result, or would expect it in some number of milliseconds. But N_b is not the only node with a schedule to keep. N_a has its own set of tasks to perform, and cannot go about rearranging its schedule just to accommodate N_b. Especially since N_a has no reason to believe that it is not N_b which is running fast.

Synchronization prevents these types of problems since synchronization allows, among other things, the coordination of schedules. Traffic lights need only be synchronized to within a second or two. Even an inexpensive network has delay times on the order of tens of milliseconds, so synchronizing traffic signals only requires that they be connected in some fashion. Multicomputers, especially real-time multicomputers, often must be synchronized within *microseconds*, while the time for a signal to travel from one node to another may be on the order of milliseconds. Faulty nodes, clocks, and communications systems must also be dealt with. And, the price of failure may be far more than a traffic jam.

This dissertation considers some of the difficulties of synchronization in distributed real-time multicomputers. A new classification scheme is presented for synchronization algorithms which breaks them into three parts: distribution of clock information, estimation of clock skews, and adjustment of clock values. Two new adjustment algorithms are presented which can greatly reduce the difficulty of synchronizing large systems. A new and highly-efficient clock distribution algorithm is presented which, along with two provided probabilistic estimation algorithms, allows highly accurate skew estimation even in large systems. Finally, an algorithm for continuously distributing clock information is presented, the only such algorithm known that does not require special hardware. Continuous distribution of clock information allows nodes to continuously monitor their skews with respect to the rest of the system, and thus maintain a lower average skew between nodes. Put together, these algorithms make an excellent synchronization algorithm in its own right. But taken separately, they can be combined with algorithms from other sources to custom design a synchronization algorithm for almost any need.

The organization of the remainder of this dissertation is as follows. Chapter 2 presents the new classification scheme for synchronization algorithms, describes previous work in

the field, and defines notation used in the rest of the dissertation. Chapter 3 describes a fault-tolerant clock adjustment algorithm which relaxes many of the restrictions found in similar adjustment algorithms, especially with regard to large systems. Chapter 4 describes a clock adjustment algorithm which greatly reduces the number of estimates each node must make. Chapter 5 describes a clock distribution algorithm and two probabilistic estimation algorithms which allow accurate, efficient estimation even in large systems. Chapter 6 describes a clock distribution algorithm which operates "continuously", allowing continuous monitoring of clock skews. Chapter 7 contains a summary and proposals for future work.

CHAPTER 2

SYNCHRONIZATION

Synchronization is generally achieved by synchronizing *clocks*. A clock is a counter, driven by a known frequency, so it increments at a known rate. If all nodes have their own clocks, incrementing at nearly identical rates, having approximately the same values at the same time, the clocks can be used as a global time base. Synchronization is achieved if there is a bound on the maximum skew between clocks.

The nemesis of synchronization is the variability of clock rates. No two clocks run at exactly the same rate. Over time, the difference in clock rates leads to an increase in the difference between clock values (called the *skew*). Synchronization becomes unnecessary if all nodes share the same clock. A single global clock is generally impractical due to fault-tolerance considerations and possible delays and difficulty in reading such a clock. An external time reference may also be used. UTC (Universal Time Coordinated) [34] is available via telephone, radio, or satellite, from several sources, at varying levels of accuracy [4, 10, 45, 47]. However, equipping each node so it can read UTC may be an expensive proposition in terms not only of money, but size, weight, and power consumption as well. And some systems must operate where UTC is not available (e.g., spacecraft or military systems). Another option is to reduce the variability in clock rates. Atomic clocks [12, 43, 44] and temperature-compensated quartz oscillators are far more accurate frequency sources than the simple quartz oscillators found in most computer systems, and can greatly reduce or eliminate the need for synchronization (except for an initial synchronization). Again, such devices can add considerably to system cost, size, weight, and power consumption.

While the above solutions attempt to reduce or eliminate variability in clock rates, synchronization algorithms attempt to compensate for it. Synchronization algorithms are generally cheaper, and often better than the above solutions. The synchronization algorithm at each node communicates with the synchronization algorithms at other nodes and tries to

maintain a bound on the skew between the local clock and every other clock in the system. Most synchronization algorithms run periodically, and alter the clock value directly. Some run continuously, and usually attempt to alter the clock rate itself. A few are hybrids, and combine periodic adjustments of the clock value with modification of clock rates.

The remainder of this chapter starts by introducing a new and simple classification scheme for synchronization algorithms, and by giving a brief overview of known algorithms. This is followed by a discussion of the focus of this dissertation, which specifies what areas of synchronization and types of synchronization algorithms will be dealt with, and why. The chapter concludes by introducing some definitions, assumptions and notation that will be used throughout this dissertation.

2.1 Parts of Synchronization

In order to better understand synchronization algorithms, it is helpful to break them up into three separate operations, and classify them according to the type of algorithm employed for each operation. The three operations that make up synchronization algorithms are as follows:

1. Distributing clock information.
2. Estimating clock skews.
3. Adjusting clock values.

The algorithms which perform these operations are referred to as the clock distribution, estimation, and adjustment algorithms, respectively. Interaction between these operations is only through input and output, i.e., the output of one operation is the input to another. One may therefore assume that the operations proceed sequentially, in the order listed above, and that the algorithms for each operation may be considered separately. The only dependencies between the algorithms for each operation is the requirements they place on one another, e.g., an estimation algorithm may require the clock distribution algorithm distribute clock information in a particular format.

A number of algorithms have been proposed for each operation, but they generally fall into a few general classes. The rest of this section presents the various classes of algorithms for each operation, and briefly points out the advantages and disadvantages of each.

2.1.1 Distributing Clock Information

Distributing clock information is the first operation, and the most resource intensive. The efficiency and accuracy with which it is done has a direct effect on the success of the remaining two operations. Two general classes of algorithms exist, distinguished by whether or not special hardware is required.

Hardware Distribution

Hardware clock distribution algorithms require a special network dedicated solely to the transmission of synchronization information. Each node uses this network to transmit its clock signal, and to receive the clock signals of other nodes. The clock information transmitted is not a clock value, but rather the output of the oscillator which drives the clocks of each node.

In most cases [16, 20, 21, 42], the synchronization network must be completely connected. This requires on the order of n^2 communications lines in an n-node system. This results in a rapid increase in cost as system size increases. In [39] a hardware distribution method is presented which does not require a completely-connected synchronization network, however the number of network connections is still on the order of n^2.

Synchronization algorithms which use hardware clock distribution algorithms usually operate continuously, and place a very small bound on maximum skew. However, they also require specialized estimation and adjustment algorithms not used with other clock distribution algorithms (although there are parallels). The estimation algorithms consist of comparing the phases of the incoming clocks and selecting one of the signals as a reference. The selected signal is usually some sort of median of the available signals [21, 42]. The adjustment algorithms consist of using a phase-locked loop to synchronize the local clock signal to the selected reference. The authors of [40] point out that, especially in larger systems, the signal propagation delay may vary greatly due to variation in line lengths, parasitic capacitances and inductances, etc. This may cause signals to arrive "out of order", and affect which signal is chosen as a reference. A solution is given in [40], but it doubles the number of network connections.

An alternative to the above would be to use each of the incoming signals to drive a counter. The counters then show the current clock values of the nodes supplying the oscillator signals (with some allowance for signal propagation delay). The estimation algorithm then consists of subtracting the local clock value from each of the counters. Any of the

adjustment algorithms of either Section 2.1.3, Chapter 3, or Chapter 4 can be used for the adjustment algorithm, and each node must tell the rest of the nodes how much it has adjusted its clock so that they may update their counters. There is no evidence that this approach has never been tried. Perhaps because it has the cost of hardware clock distribution algorithms, but cannot bound the skew as tightly as the above algorithms. However, it does allow for much greater flexibility in the choice of adjustment algorithm.

There is another type of hardware distribution algorithm that does not require a completely connected synchronization network, but is intended primarily for synchronization of a switching network, such as a telephone transmission system [1, 5, 18, 25, 26, 33]. These algorithms are used to establish a common frequency between switches so that transmitted data is not lost. They are not perfect however, and occasional *slips* [1] cause loss of data. Skew between clocks driven by these frequencies will still increase, albeit rather slowly. It may be possible to use one of these algorithms to control the frequencies, and use a periodic synchronization algorithm to bound the skew. Such an arrangement would not bound the skew as tightly as algorithms using standard hardware clock distribution algorithms, but would be considerably cheaper to implement.

Network Distribution

Network distribution algorithms use the existing communications network to transmit clock values. Special messages, called *synchronization messages*, are the carriers of this information. In most cases the synchronization messages are sent in some type of broadcast [14, 15, 22, 27, 35, 36, 41]. Other algorithms use private communication between nodes [3, 11]. Synchronization algorithms which use network clock distribution algorithms are usually periodic, since continuous network traffic would likely interfere with other system operations.

The principal advantages of network clock distribution algorithms are their low cost, and their great flexibility. No extra hardware is required, the only cost is in the network traffic generated. Flexibility comes from using the system network. Unlike hardware clock distribution algorithms, which distribute only oscillator signals, network clock distribution algorithms can distribute any type of information which can be carried by the system network. This allows the clock distribution algorithm to be tailored to distribute information of the type, and in the form required by the estimation algorithm. This allows network clock distribution algorithms to be used with a wide variety of estimation and adjustment

algorithms. The estimation and adjustment algorithms in the remainder of this section essentially assume a network clock distribution algorithm is used.

The principal drawbacks of network clock distribution algorithms are the network load they generate, and the uncertain effects network delay has on the clock information they distribute. The broadcasts of synchronization messages can cause a noticeable network load and interfere with other system operations. The periodic nature of these broadcasts may force time-critical tasks to schedule communication around the synchronization algorithm. Perhaps more important is the effects of network delay. Synchronization messages arrive at their destinations after some unknown and non-trivial delay. Clock information ages quickly, and clock information of uncertain age is of little use to any estimation algorithm. The net result is poorer estimates, and a much larger bound on clock skew than with hardware clock distribution algorithms. The algorithm in [35] is something of a hybrid between hardware and network distribution. Synchronization messages are sent on the communications network, but special hardware is required at each node to measure communications delay. This eliminates the problem with delays, at some extra cost in hardware.

2.1.2 Estimating Clock Skews

The clock information that each node receives is used to estimate the skew of the local clock with respect to the rest of the clocks in the system. Each estimate has an associated *uncertainty*, which is the maximum amount by which the estimate may be in error. The uncertainty of the estimates is a limiting factor in the synchronization algorithm. The higher the uncertainty, the more difficult it is to maintain tight bounds on skew.

There are two general types of estimation algorithms, divided according to the uncertainty of their estimates.

Simple Estimation

Simple estimation algorithms compute the skew as the difference between the local clock value when a synchronization message was received, and the clock value on the synchronization message, minus some constant [24, 22, 27, 30, 35]. The constant value is an attempt to account for the communications delays. Usually, either the minimum or mean network delay is used.

Simple estimation algorithms are easy to implement, and require a minimum of processor time to execute. They also place few demands on the clock distribution algorithm, requiring

only a single clock value from every node for which an estimate is to be made.

The principal drawback of absolute estimation algorithms is the high uncertainties they generate. In most cases the uncertainty of estimates is determined by the maximum variability in network delay, which in a large or busy network may be quite high indeed. [35] manages to avoid these difficulties by using special hardware to track of network delay, but adds extra cost to each node.

Probabilistic Estimation

Probabilistic estimation algorithms make assumptions about the distribution of network delays, and use this information to improve their estimates [2, 11, 36]. They are generally repeated executions of simple estimation algorithms where the multiple estimates for a single node are combined.

Probabilistic estimation algorithms can, in theory, produce estimates with as low an uncertainty as is desired. They allow synchronization algorithms to overcome the primary disadvantage of network clock distribution algorithms: unpredictability of network delays.

The principal drawback of probabilistic estimation algorithms is their need for large quantities of clock information. This is a result of the repeated executions of the simple estimation algorithm, each execution requiring a new clock value. Often this causes the clock distribution algorithm to be run repeatedly over a specified time interval [2, 11, 32] Another approach is found in [36], where a special broadcast is described which "pipelines" multiple broadcasts.

2.1.3 Adjusting Clock Values

Adjusting clock values is the ultimate goal of the synchronization algorithm, and always the last of the three operations to be performed. The adjustment algorithm takes the estimates generated by the estimation algorithm and produces a *synchronization adjustment* which, when added to the local clock, will synchronize it with the rest of the clocks in the system. The ability of the adjustment algorithm to synchronize the system is generally limited by the uncertainty of the estimates.

Adjustment algorithms have been the focus of most of the research in synchronization. There are a number of them, and they fall into two general categories: master/slave algorithms, and peer algorithms.

Master/Slave Algorithms

Master/slave algorithms assign one or more nodes to the role of "master clock", and have all the "slaves" synchronize to the masters [2, 11, 32]. Master clocks are usually given either a highly accurate frequency source, such as an atomic clock, or have access to some external time standard, such as UTC. Indeed, synchronizing a system to an external time standard is what master/slave algorithms are best suited for.

Master/slave algorithms are simple in concept, and greatly reduce the number of estimates that must be made, thereby reducing the work of the clock distribution algorithm.

The principal disadvantage of master/slave adjustment algorithms is the complexities involved in providing fault-tolerance. More than one master must be present if failures of the master clock, or the node containing the master clock, are allowed. This increases the cost as multiple nodes must be outfitted with the extra hardware needed for a master clock. The adjustment algorithm loses its surface simplicity as nodes must be able to detect or mask faulty masters. Finally, if the masters do not use an external time standard, the masters must synchronize themselves. Synchronization amongst masters may be done with a peer adjustment algorithm, or with a master/slave algorithm, forming a hierarchy of masters, as in NTP [32].

Peer Algorithms

In peer adjustment algorithms all clocks are on an equal footing. They bear a considerable resemblance to many distributed agreement algorithms, only in this case the value which is being agreed upon changes with time. There are three general types of peer adjustment algorithms.

Agreement algorithms are direct applications of distributed agreement algorithms used for clock adjustment [14, 15, 41]. In agreement algorithms the nodes simply agree that the current time is T, and all nodes set their clocks to T. The synchronization adjustment is the difference between the current clock value and T. Since in a synchronized system the clocks of all non-faulty nodes should be approximately T anyway, this brute-force approach works reasonably well. The simplicity of agreement algorithms is offset by limits on how tightly skew can be bounded. The bound on maximum skew is determined by the maximum run-time of the agreement algorithm, which can be rather large.

Consistency algorithms compute the synchronization adjustment as a function of the estimates, usually the median. These algorithms are simple to implement, and the bound

on skew is determined largely by the uncertainty of the estimates. The primary drawback of consistency algorithms is that they require that any two non-faulty nodes agree in their estimate of any other node (faulty or not) [22]. In other words, Byzantine behavior [23], where a faulty node gives inconsistent information, is not allowed. While there are algorithms to detect and mask Byzantine behavior, they are complicated and induce considerable overhead. For this reason, consistency algorithms are rarely used.

Convergence algorithms compute the synchronization adjustment as a function of a subset of the estimates, discarding some estimates in the hopes of eliminating the effects of faulty nodes. Perhaps the simplest and most obvious example is to average the estimates [22]. Large estimates are discarded so that faulty nodes may not overly influence the computation. Other adjustment algorithms of this type select one estimate to use as the synchronization adjustment. In [27] the largest and smallest m estimates are discarded, where m is the maximum number of faults, and the median of the remaining estimates is used as the synchronization adjustment. In [35] the m smallest estimates are discarded, and the smallest remaining estimate is used as the synchronization adjustment. Convergence algorithms are simple to implement, and the bound on skew is largely determined by the uncertainty of the estimates. Byzantine behavior is tolerated, as long as the number of faults does not exceed the specified maximum. These attributes have made convergence algorithms the most popular type of peer adjustment algorithm.

2.2 Focus

As may be seen from the previous section, while synchronization is considered an important problem, relatively little work has been published. A few dozen papers contain all important contributions to the field. Familiarity with the field provides a possible answer, while the problem seems simple at the outset, it really is quite difficult. There are no simple, efficient, elegant algorithms that work in a wide variety of circumstances.

The focus of this dissertation is not to provide such an algorithm. It is unlikely that one exists. Instead, the classification system of the previous section is exploited. It is noted that the different operations which compose synchronization are independent to some degree. It is therefore possible to create different synchronization algorithms by mixing and matching different algorithms for the different operations. The focus of this dissertation is to develop algorithms for the various *operations* of synchronization, and to make them flexible enough

that they can work not only with one another, but with other algorithms as well.

Certain types of algorithms will be of particular interest:

Network algorithms: The cost of hardware algorithms is too high for all but the smallest systems. There are also significant problems with using hardware synchronization when the system is distributed over a wide geographic area.

Probabilistic algorithms: Probabilistic algorithms have the potential to produce estimates with arbitrarily low uncertainties. Their major drawback is the network load they produce. This is becoming less of a problem as networks become faster and bandwidth increases, especially if an efficient clock distribution algorithm is available.

Peer algorithms: Master/slave algorithms appear simpler on the surface, but that simplicity disappears when one considers problems of detecting master failures and synchronizing masters. Peer algorithms do not have such difficulties, and often offer better fault-tolerance since they do not depend on any master clocks.

Chapters 3 and 4 introduce two peer adjustment algorithms that offer significant improvements over existing algorithms. Chapters 5 and 6 then introduce two new network distribution algorithms, and two probabilistic estimation algorithms that work with them.

2.3 Notation and Assumptions

The distributed systems under consideration are assumed to be *multicomputers*. Multicomputers are defined to be distributed systems where the interaction and cooperation between nodes is assumed to be very high, high enough that the system can often be seen as a single computer. This close cooperation is the principal reason for synchronization. In order to facilitate communication and task allocation, multicomputers usually have a homogeneous network topologies. The most common examples are the hypercube [38], the torus [29], and the hexagonal mesh [9]. These regular topologies will be exploited when possible.

For the purposes of synchronization, only those nodes which have their own clock are of interest, nodes without clocks are important only in the way that they affect message delivery time. The system is assumed to have n nodes with clocks, denoted $N_0, \ldots, N_{n-1}$. Should the system have a different node numbering, or if only some nodes have clocks, the node numbers can be re-mapped accordingly. Each node's clock runs at its own rate, and

each node has a *clock process* which maintains and periodically adjusts the clock value, and works with the synchronization algorithm to keep the clock synchronized with the rest of the clocks in the system. The clock process may be implemented either in software, or in special hardware (similar to the "clock chips" currently available). The current clock value (and other values associated with the clock) may be obtained by querying the clock process, or by accessing its registers directly.

2.3.1 Faults

In this dissertation faults are considered only so long as they affect the synchronization algorithm. These faults are divided into two types, those that affect clocks, and those that affect nodes.

Definition 2.1 *A faulty clock is any clock which either increments by values other than 1, or increments at a which differs by more than ϱ from its specified rate.*

The value of ϱ is specified by the manufacturer of the frequency source used for the clock, and is called the clock's *maximum drift rate.*

Definition 2.2 *A faulty node is any node which loses, corrupts or otherwise alters the contents of synchronization messages, or includes incorrect or misleading information in synchronization messages.*

Note that faulty nodes are defined only in terms of what they do to synchronization messages. Since faulty clocks are generally not distinguishable from faulty nodes, the term faulty node will be used to refer both to faulty nodes, and to non-faulty nodes which contain faulty clocks.

2.3.2 Synchronization Parameters

There are two primary system parameters which affect the operation of the synchronization algorithm:

m: The maximum number of faulty nodes that are present in the system at any time. Synchronization becomes more difficult as m is increased.

δ: The *target* skew. If the system is synchronized the clocks of any two non-faulty nodes differ by no more than δ. Obviously, as δ is reduced synchronization becomes more difficult and consumes more system resources.

2.3.3 Clocks

A *clock function* is a non-decreasing function mapping from an interval of the real numbers into the integers. The *clock* for N_i, $C_i(t)$, is a clock function from some real-valued infinite-precision external Newtonian time frame into the set of integers. This external time frame is the same for all clocks, and can be assumed to be TAI, UTC or some equivalent standard. Throughout this dissertation lower-case letters are used for times in the external frame, and capital letters are used for clock values.

The value of $C_i(t)$ depends on three other functions:

C_i^R: The *raw* clock. The raw clock corresponds to the "clock" on most systems. It is a counter which is incremented at regular intervals. $C_i^R(t)$ is a clock function, and therefore may never decrease.

C_i^A: The *clock adjustment*. The clock adjustment is added to the raw time to get the clock value, i.e., $C_i(t) = C_i^R(t) + C_i^A(t)$. $C_i^A(t)$ is not a clock function, and it may decrease as necessary to maintain synchronization.

$\rho_i(t)$: The clock *drift rate*. The drift rate is an instantaneous measure of the current difference between the rate $C_i^R(t)$ increments, and the rate a "perfect" clock, $\lfloor t \rfloor$, increments. If $C_i^R(t_0) = T_0$, then $C_i^R(t_1) = T_0 + \int_{t_0}^{t_1}(1 + \rho_i(t))dt$. The drift rate is bounded by the maximum drift rate, ϱ, i.e., $|\rho_i(t)| \leq \varrho \, \forall t$.

Both C_i^R and C_i^A are maintained by the clock process at N_i. The value of $\rho_i(t)$ is generally not known, while ϱ is a system parameter.

There is a fourth function maintained by the synchronization algorithm and monitored by the clock process:

C_i^T: The *target adjustment*. The target adjustment is what the synchronization algorithm has determined the value of $C_i^A(t)$ *should be*. Like the clock adjustment, the target adjustment may either increase or decrease, and is therefore not a clock function. The clock process monitors $C_i^T(t)$, and adjusts C_i^A so that $C_i^A(t) = C_i^T(t)$. However, since C_i must be a clock function, the clock process must gradually change the value of $C_i^A(t)$. In particular, the clock process can reduce C_i^A only as fast as C_i^R increases.

The *synchronization clock*, C_i^S, is the sum of the raw clock value and the target adjustment, i.e., $C_i^S(t) = C_i^R(t) + C_i^T(t)$. Because the target adjustment, unlike the clock

adjustment, is changed in sudden large increments, the synchronization clock is not a clock function.

Of the three "clocks" at each node, C_i, C_i^R, and C_i^S, only C_i^R and C_i^S are of interest to the synchronization algorithm. The relatively constant rate of C_i^R makes it good for measuring elapsed time, and the function of the synchronization algorithm is to synchronize C_i^S and C_j^S for all node pairs N_i and N_j, $i \neq j$.

For notational convenience, the above functions will be referred to by name only, e.g., C_i^R instead of $C_i^R(t)$, when the time in the external frame is understood (usually the current time).

2.3.4 Skew

The *skew* between two time-dependent values is the current difference in their values. Skew could refer to any two values, but in the context of synchronization it usually refers to clock values. In particular, the skew of N_i with respect to N_j, α_{ilj}, is understood to be the current difference in the raw clocks of N_i and N_j, i.e., $C_i^R - C_j^R$.

Synchronization algorithms measure skew between raw clock values because there is a specified minimum and maximum rate at which they are incremented. Therefore, there is a specified minimum and maximum rate at which the skew changes. This is very important to the synchronization algorithm, which tries to estimate the skew, which changes over the course of the synchronization algorithm. A bound on the rate the skew changes allows the synchronization algorithm to account for the change.

2.3.5 Timestamps

A *timestamp* is a data structure used to carry information about clock values through the system. There are three different types of timestamps commonly used in systems, and they are distinguished by the different what values they carry.

Raw timestamp: The current raw clock value.

Adjusted timestamp: The current values of both the raw clock and the clock adjustment, listed separately.

Synchronization timestamp: The current values of both the raw clock and the target adjustment, listed separately. Listing the raw clock and target adjustment separately instead of summing them allows other nodes to track the values of both functions.

This allows nodes to know one another's current target adjustments, and allows them to estimate the skews between their respective raw clocks.

Synchronization algorithms use raw and synchronization timestamps only. Many systems provide facilities to automatically add raw timestamps to messages. Few provide the means to automatically add synchronization timestamps. However, the target adjustment is under the control of the synchronization algorithm, and is adjusted infrequently. The synchronization algorithm may therefore generate synchronization timestamps for its messages by adding the current value of the target adjustment to its messages, and letting the automatic timestamping feature complete the process. However, the synchronization algorithm must be sure that the target adjustment will not be changed until after the message is sent.

CHAPTER 3

FAULT-TOLERANT CLOCK ADJUSTMENT

As the final phase of synchronization, the needs of the adjustment algorithm are what drive the development of the clock distribution and estimation algorithms. For this reason, clock adjustment is considered first.

The purpose of the adjustment algorithm is to compute a *synchronization adjustment* which is then added to the target adjustment. The synchronization adjustment is the adjustment algorithm's estimate of how far off the local clock is from either the master clock(s) (if master/slave synchronization is used), or from the rest of the system (if peer synchronization is used.) By adding it to the target adjustment the synchronization algorithm effectively informs the clock process about how much it must either speed up or slow down the system clock in order to keep it synchronized.

Adjustment is a simple procedure in master/slave synchronization algorithms. A slave makes skew estimates of each master. Of the estimates obtained the largest consistent set is chosen (i.e., estimates do not differ by more than their uncertainties plus whatever skew is allowable between masters), the synchronization adjustment is the element of this set with the lowest uncertainty. There are several possible variations on this procedure. Slaves may only attempt to estimate a single master, or a subset of the masters. Each slave may be assigned a particular master whose estimate is used as its synchronization adjustment (as long as its skew is consistent), regardless of whether or not the estimate for that master has the lowest uncertainty. Or, an average of the consistent set may be used.

With peer synchronization, the difficulty is finding a function that all nodes can compute which, in spite of current differences between their clocks and the resulting differences in skew estimates, will nevertheless yield consistent results. Interactive convergence [22] is one of the most commonly-used adjustment algorithms. This is perhaps due to its intuitive nature and simplicity, it simply averages the estimates. However, interactive convergence

assumes that all estimates have the same uncertainty, and that estimates will be available for any non-faulty node. While these assumptions are reasonable in a completely-connected network, they are somewhat restrictive in the more general case. Usually, the cost of making an estimate increases both as the distance to the node increases, and as the uncertainty decreases. It may thus be impractical to get estimates of distant nodes with the same uncertainty as those of nearby nodes, and it is wasteful to assume the estimates of nearby nodes have the same high uncertainty as those of distant nodes. Similarly, faults, transient network loads, and other problems may make it impossible for a non-faulty node to make estimates of some other non-faulty nodes (at least temporarily). Furthermore, interactive convergence tolerates up to $n/3$ faulty nodes. Most systems would fail long before so many nodes have failed, and it seems silly for a failed system to maintain synchronization.

This chapter shows how these restrictions of the interactive convergence algorithm can be relaxed. Nodes are allowed to consider the uncertainty of individual estimates, and take advantage of the low-uncertainty estimates available. Nodes are also not bound to make estimates of every other non-faulty node, as long as each is able to make some minimum number of estimates. The number of faulty nodes is specifiable instead of being fixed at $n/3$. Finally, techniques are introduced to handle cases where a single fault may alter a non-faulty node's estimates of other non-faulty nodes.

The chapter starts by introducing the parameters that govern the adjustment algorithm. Then, selection of estimates is described. Computations are done to determine the minimum number of estimates needed to maintain synchronization under the assumption that estimates of non-faulty nodes are correct. Finally, the assumption of correct estimates of non-faulty nodes is lifted, and the results analyzed.

3.1 Adjustment Parameters

There are three principal parameters that control the adjustment algorithm. Selection of these parameters involves a number of trade-offs with the estimation algorithm, and careful selection is necessary to ensure the synchronization algorithm does its job without overly loading the system.

ε: The maximum tolerable uncertainty in the synchronization adjustment. The value of ε must be strictly less than δ.

r: The maximum time between re-synchronizations. A system will not normally synchronize itself constantly, there is a minimum time needed to distribute clock information and to make estimates, and the cost is too great. Instead the synchronization algorithm runs periodically, with a period no greater than r.

τ: The maximum skew between two nodes which have just adjusted their clocks. Synchronizing the clocks of the system only counters the effects of clock drift, instead of preventing it. Immediately after synchronization the clocks will start drifting apart again. Since skew cannot be allowed to become greater than δ, and the synchronization algorithm runs only periodically, the synchronization algorithm must make sure that immediately after synchronization the maximum skew is less than $\tau < \delta$, where $\delta - \tau$ is large enough to absorb any clock drift that may occur before the synchronization algorithm runs again.

The value of ε determines how difficult the job of the estimation algorithm is. The lower ε, the lower the uncertainty of the estimates needed by the adjustment algorithm, and the more synchronization messages the estimation algorithm has to send. On the other hand, the lower ε, the fewer estimates the adjustment algorithm needs in order to compute the synchronization adjustment, and the fewer low-uncertainty estimates the estimation algorithm has to produce.

The difference between δ and τ determines the maximum value of r. The maximum drift rate of any clock is ϱ, so the maximum drift rate between any two clocks is 2ϱ, and, the relationship between r and τ is as follows:

$$r \leq \frac{\delta - \tau}{2\varrho} \tag{3.1}$$

Reducing τ increases the difference between δ and τ, allowing clocks to drift further before they must be re-synchronized. It therefore increases r, and since re-synchronizations happen less often, the time spent synchronizing is reduced. Reducing τ also makes each synchronization more costly, either ε or the maximum tolerable faults must be reduced, or the number of estimates needed must be increased.

Interdependence between these three parameters can make selecting their values tricky. However, since the synchronization algorithm should interfere as little as possible with system operation, the percentage of time the synchronization algorithm can run will often be bounded. Given this bound and the expected run time of the clock distribution and

estimation algorithms, one can find a lower bound for r, and using Equation (3.1) provides an upper bound for τ. The value of ε is then chosen in accordance with the ability of the estimation algorithm to produce the necessary number of low-uncertainty estimates.

3.2 Selecting the Estimates

Adjustment begins with the skew estimates. The estimation algorithm provides a skew estimate for every node for which an estimate is available, regardless of the the uncertainty. Of the set of estimates the algorithm selects the largest subset E such that all the skews in E are within certain bounds, and the sum of the uncertainties of the estimates in E is less than $e\varepsilon$, where $e = |E|$. Exactly what bounds the elements of E must meet is the distinguishing characteristic of the various adjustment algorithms discussed in this Chapter, and is discussed in Sections 3.3 and 3.4. The estimates in E are the *accepted* estimates. The average of the accepted estimates is the synchronization adjustment.

As the average of the accepted estimates, the synchronization adjustment has an uncertainty less than or equal to ε. But that uncertainty is with respect to the average of the actual skews of nodes with accepted estimates, not to the average of the actual skews of all non-faulty nodes. This difference becomes more crucial as the number of accepted estimates decreases. When the number of accepted estimates per node is sufficiently small it is possible for a *clique* of nodes to form, where each member of the clique only accepts estimates of other nodes in the clique. Such a clique is not influenced by the clock values of nodes outside the clique, and thus is free to drift arbitrarily far from the rest of the system. Clearly, there must be some minimum number of accepted estimates that a node must have in order to add the computed synchronization adjustment to the target adjustment. Its value must be greater than $n/2$ in order to prevent clique formation. This minimum is denoted ζ, and if $e \geq \zeta$, the computation of the synchronization adjustment is said to be *successful*, and it is added to the target adjustment. Otherwise it is discarded.

3.3 Clock Adjustment With Reliable Estimates

While the adjustment algorithm must be able to tolerate *some* faults, it cannot possibly tolerate all possible faults. In particular, it cannot be expected to work if the estimates provided by the estimation algorithm are in no way related to the actual skews. Specifically, estimates of non-faulty nodes must be *reliable*. Estimates of non-faulty nodes are reliable

if the difference between the actual skews and the estimated skews is always less than their respective uncertainties (with appropriate allowances made for probabilistic estimation algorithms). Seen from a different perspective, no faulty node can cause a non-faulty node to incorrectly estimate another non-faulty node.

Estimates of faulty nodes, on the other hand, are not reliable, since a faulty node has no defined clock value. Moreover, a faulty node, N_f, is assumed to be able to manipulate the estimation algorithms of any non-faulty nodes, N_i and N_j, into computing whatever estimates of N_f that are desired. The estimates need not even be in agreement, the difference between N_i's and N_j's estimate of N_f need not be equal to the skew between them, α_{ij}.

3.3.1 Calculating ζ

Calculating ζ for given values of δ, m, ε, and τ is relatively simple. The first step is to find the maximum skew between any two nodes immediately after they have adjusted their clocks. This value depends on δ, m, ε, and the number of estimates accepted by each node. The maximum skew must also be less than or equal to τ. The resulting inequality is re-arranged to produce a minimum for the number of estimates accepted by each node.

The most difficult step in this process is finding the maximum skew between any two nodes immediately after they have adjusted their clocks. The system is assumed to be synchronized so that the skew between any two non-faulty nodes is less than or equal to δ before any clock adjustment is done. Let N_i and N_j be any two nodes in the system. Without loss of generality, assume $C_j^S \geq C_i^S$. Let N_i accept $e_i \geq \zeta$ estimates, and N_j accept $e_j \geq \zeta$ estimates. The goal is to find, for all possible values of e_i and e_j, the maximum skew between N_j and N_i after they have adjusted their clocks. The maximum skew increases as e_i and e_j decrease, so ζ is the smallest integer such than whenever *both* e_i and e_j are greater than or equal to ζ, the maximum skew is less than or equal to τ.

The skew between N_j and N_i after they have adjusted their clocks is equal to the skew between them before they adjusted their clocks, plus the difference between the synchronization adjustments of N_j and N_i. Finding the maximum is not as simple as finding the maximum of N_j's synchronization adjustment and the minimum of N_i's synchronization adjustment. The two synchronization adjustments depend on one another as well as the skew between N_j and N_i before they adjust their clocks. However, there are certain conditions that *must* hold for the skew between N_j and N_i after they have adjusted their clocks to be the maximum for the given values of e_i and e_j:

1. N_i's estimates are, on average, ε too low, while N_j's estimates are, on average, ε too high. What this means for estimates of non-faulty nodes is clear, if the uncertainty for a particular estimate is ϵ, then the estimate is either ϵ below (for N_i) or above (for N_j) the actual skew value. What this means for estimates of faulty nodes is not so clear. Since each node establishes bounds within which every accepted estimate (plus or minus its uncertainty) must lie, estimates of faulty nodes are assumed to lie just within these bounds. That is, if the uncertainty of N_i's estimate of a particular faulty node is ϵ, N_i's estimate for that node will be ϵ less than N_i's lower bound on accepted estimates.

2. Given any non-faulty node, either N_i, N_j, or both accept an estimate for that node. This implies that the set of nodes for which both N_i and N_j accept estimates has the fewest possible members, which is $e_i + e_j - n$.

3. The nodes for which only N_j accepts estimates will have skews of δ with respect to the nodes for which only N_i accepts estimates.

4. The number of faulty nodes is m, and *both* N_i and N_j accept estimates of every faulty node.

5. Faulty nodes have no specific clock values. Two non-faulty nodes may therefore estimate very different and inconsistent skews for the same faulty node. Thus, N_i's skew estimates for the faulty nodes are the smallest estimates N_i can accept, and N_j's skew estimates for these same faulty nodes are the largest estimates N_j can accept.

The reasoning behind the above conditions is clear if one keeps in mind that the general objective is to maximize N_j's synchronization adjustment while minimizing N_i's synchronization adjustment. It is then easy to verify that if any of the above conditions does not hold, then the skew between N_j and N_i after they have adjusted their clocks will not be the maximum possible.

The following definitions are then useful in the computation of ζ. Note that in these definitions all skews are the *actual* skews instead of the estimated skews, the difference between actual and estimated skew will be accounted for by including a term for maximum uncertainty.

$\underline{\gamma_i}$ and $\overline{\gamma_i}$: The minimum and maximum skews (with respect to N_i) of non-faulty nodes whose skew is estimated by N_i. Note that since a node will automatically accept an

estimate of its skew with respect to itself (0), $\underline{\gamma_i}$ has a maximum of 0, and $\overline{\gamma_i}$ has a minimum of 0. Also, $\overline{\gamma_i} - \underline{\gamma_i} \leq \delta$.

$\underline{\gamma_j}$ and $\overline{\gamma_j}$: The minimum and maximum skews (with respect to N_j) of non-faulty nodes whose skew is estimated by N_j. Again, the maximum of $\underline{\gamma_j}$ is 0, the minimum of $\overline{\gamma_j}$ is 0, and $\overline{\gamma_j} - \underline{\gamma_j} \leq \delta$.

Γ_i: The sum of the skews (with respect to N_i) of the $n - e_j$ non-faulty nodes whose skew is estimated by N_i only.

Γ_j: The sum of the skews (with respect to N_j) of the $n - e_i$ non-faulty nodes whose skew is estimated by N_j only.

Γ_{ij}: The sum of the skews (with respect to N_i) of the $e_i + e_j - n - m$ non-faulty nodes whose skew *both* N_i and N_j estimate.

The value of ζ is a function of n, m, τ, and ε. The exact form of the function is determined by the bounds placed on accepted estimates.

3.3.2 Unrestricted Range Mean

The obvious approach to bounding the accepted estimates, and the one used in [22], is to accept only those estimates where the absolute value of the estimate, minus its uncertainty, is less than or equal to δ. The rationale is simple, if the system is currently synchronized all non-faulty nodes will be within δ of one another, and if the local node is non-faulty and estimates are reliable, the absolute value of the skew estimate for any non-faulty node must be less than δ (allowing for any uncertainty in the estimate). If any estimates are too large, either they come from a faulty node or the local node is faulty. This approach is easy to do, and limits the range of estimates to $\pm\delta$. It is called *unrestricted* range mean because the only restriction on the range of the estimates is the one enforced by magnitude restrictions, i.e., two estimates could differ by 2δ even though such a situation is not possible for non-faulty nodes.

The value of ζ is found using the procedure in Section 3.3.1. Since C_j^S is assumed to be larger than C_i^S, the skew after synchronization will be the current skew (α_{jli}) plus the synchronization adjustment of N_j minus the synchronization adjustment of N_i. Maximizing this value means minimizing the synchronization adjustment of N_i, while maximizing the synchronization adjustment of N_j. As discussed in Section 3.3.1, N_i's estimates of the faulty

nodes should be as small as possible, $-\delta$ minus their uncertainties, while N_j's estimates of the faulty nodes should be as large as possible, δ plus their uncertainties. To simplify matters, the individual estimate uncertainties are combined into two terms, $-e_i\varepsilon$ for N_i, and $e_j\varepsilon$ for N_j. The following equations result:

$$\begin{aligned} \tau &> \alpha_{jli} + \frac{1}{e_j}\left(\Gamma - ij - (e_i + e_j - n - m)\,\alpha_{jli} + \Gamma_j + m\delta + e_j\varepsilon\right) \\ &\quad - \frac{1}{e_i}\left(\Gamma - ij + \Gamma_i - m\delta - e_i\varepsilon\right) \qquad (3.2) \\ e_ie_j\tau &> (e_i - e_j)\,\Gamma - ij + e_i\Gamma_j - e_j\Gamma_i - e_i\,(e_i - n - m)\,\alpha_{jli} + (e_i + e_j)\,m\delta \\ &\quad + 2e_ie_j\varepsilon \qquad (3.3) \end{aligned}$$

The $(e_i + e_j - n - m)$ term compensates for the fact that the skews summed in Γ_{ij} are with respect to N_i, and N_j will estimate skews α_{jli} lower.

Since e_i and e_j appear on both sides of the inequality, they are considered constant for the moment. The goal is to maximize the right side of the inequality, so Γ_i should be minimized while Γ_j should be maximized. Since the minimum skew with respect to N_i of any non-faulty node is $\underline{\gamma_i}$, and given that the skew between any two non-faulty nodes is less than or equal to δ, it follows that the maximum skew with respect to N_i of any non-faulty node is $\underline{\gamma_i} + \delta$. The following inequalities then hold:

$$\begin{aligned} \Gamma_i &\geq \underline{\gamma_i}\,(n - e_j) \qquad (3.4) \\ \Gamma_j &\leq \left(\underline{\gamma_i} + \delta - \alpha_{jli}\right)(n - e_i) \qquad (3.5) \end{aligned}$$

In the worst case, the inequalities of Equations (3.4) and (3.5) are changed to equalities. Substituting into Equation (3.3) gives the following inequality:

$$\begin{aligned} e_ie_j\tau &> (e_i - e_j)\,\Gamma - ij - e_i\,(e_i - n)\,\underline{\gamma_i} + e_j\,(e_j - n)\,\underline{\gamma_i} + \underline{\gamma_i}m\alpha_{jli} \\ &\quad + (e_i + e_j)\,m\delta - e_i\,(e_i - n)\,\delta + 2e_ie_j\varepsilon \qquad (3.6) \end{aligned}$$

At this point no further reduction can be done without more information about the relative values of e_i and e_j. As a first case, assume $e_i \geq e_j$. Γ_{ij} must therefore be maximized, since the maximum skew with respect to N_i of any non-faulty node is $\underline{\gamma_i} + \delta$, the maximum value of Γ_{ij} is $\left(\underline{\gamma_i} + \delta\right)(e_i + e_j - n - m)$. Substituting into Equation (3.6) gives:

$$e_i e_j \tau \;>\; -e_i \underline{\gamma_i} + e_j \underline{\gamma_i} + e_i m \alpha_{jli} - e_j \left(e_j - n - 2m\right) \delta + 2 e_i e_j \varepsilon$$

Since $e_i \geq e_j$, the right-hand side of the above equation decreases with increasing $\underline{\gamma_i}$. It is therefore at a maximum when $\underline{\gamma_i}$ is minimized. The value of $\underline{\gamma_i}$ must be within δ of the skew of N_j, and therefore greater than $\alpha_{jli} - \delta$. Substituting gives:

$$e_i e_j \tau \;>\; e_i m \delta + e_j m \alpha_{jli} + e_j \left(n + m - e_j\right) \delta + e_i e_j \varepsilon$$

The right-hand side of the above equation increases with increasing α_{jli}, so α_{jli} should be as large as possible. The maximum of α_{jli} is δ. Substituting, and moving e_i and e_j back to the right-hand side gives:

$$\tau \;>\; \frac{1}{e_j} m \delta + \frac{1}{e_i} \left(n + 2m\right) \delta - \delta + 2\varepsilon$$

With e_i and e_j appearing only on the right-hand side of the inequality it is now possible to determine what values they can take to maximize the right-hand side. Since both are used to divide positive quantities, both should be as small as possible, i.e., both should be equal to ζ. Substituting, and solving for ζ gives:

$$\zeta \;>\; \delta \left(n + 3m\right) / \left(\delta + \tau - 2\varepsilon\right) \tag{3.7}$$

Equation (3.7) is only for the case where $e_i \geq e_j$, hence a separate derivation must be done for $e_i \leq e_j$. In this case, Γ_{ij} must be minimized, its minimum is $\underline{\gamma_i} \left(e_i + e_j - n - m\right)$. Continuing the derivation in the same way as above it turns out that $\underline{\gamma_i}$ must be maximized (0), as must be α_{jli} (δ). Again, it turns out that e_i and e_j must both be minimized (ζ), and the final result is the same as above. It follows that Equation (3.7) is the lower bound for ζ when unrestricted range mean is used to compute the synchronization adjustment.

Equation (3.7) has the properties that one would expect ζ to have. It increases with n, m, and ε. It increases as τ decreases, and becomes greater than n if τ becomes less than 2ε. Consider the case where $m = 0$, $\varepsilon = 0$, and $\tau = \delta$, a fault-free system with "perfect" estimation and "continuous" clock adjustment. Equation (3.7) reduces to $\zeta > n/2$, i.e., a nodes needs to accept estimates for just over one-half of the nodes in order to stay synchronized, exactly what would be expected in such a system.

3.3.3 Restricted Range Mean

While unrestricted range mean limits the magnitude of skew estimates, it doesn't directly limit their range. For example, a node may accept skew estimates of both δ and $-\delta$, even though it is not possible for both estimates to be of non-faulty nodes. But while one of the estimates must be of a faulty node, which one is it? As it turns out, it doesn't matter, as long as the adjustment algorithm accepts enough estimates.

With restricted range mean, a node selects the largest set of estimates whose average uncertainty is less than or equal to ε, *and* where the absolute value of the difference between any two estimates, minus their uncertainties, is less than or equal to δ. Derivation of ζ is much the same as in unrestricted range mean, only the skews of faulty nodes are different. N_i will estimate skews of $\overline{\gamma_i} - \delta$ for each faulty node (instead of $-\delta$), and N_j will estimate skews of $\underline{\gamma_j} + \delta$ for each faulty node (instead of δ). Following the example for unrestricted range mean, the following equations result:

$$\begin{aligned}
\tau &> \alpha_{jli} + \frac{1}{e_j}\left(\Gamma - ij - (e_i + e_j - n - m)\,\alpha_{jli} + \Gamma_j + \left(\underline{\gamma_j} + \delta\right) m + e_j\varepsilon\right) \\
&\quad - \frac{1}{e_i}\left(\Gamma - ij + \Gamma_i + (\overline{\gamma_i} - \delta)\, m - e_i\varepsilon\right) \\
e_i e_j \tau &> (e_i - e_j)\,\Gamma - ij + e_i\Gamma_j - e_j\Gamma_i - e_i\,(e_i - n - m)\,\alpha_{jli} + e_i m\underline{\gamma_j} - e_j m\overline{\gamma_i} \\
&\quad + (e_i + e_j)\, m\delta + 2e_i e_j\varepsilon
\end{aligned} \tag{3.8}$$

Equations (3.4) and (3.5) can again be used to eliminate Γ_i and Γ_j. Since N_i and N_j both accept estimates of some common nodes, it follows that $\underline{\gamma_j} + \alpha_{jli} \leq \overline{\gamma_i}$. Substituting into Equation (3.8) gives:

$$\begin{aligned}
e_i e_j \tau &> (e_i - e_j)\,\Gamma - ij - e_i\,(e_i - n)\,\underline{\gamma_i} + e_j\,(e_j - n)\,\overline{\gamma_i} + (e_i - e_j)\, m\overline{\gamma_i} - e_i\,(e_i - n)\,\delta \\
&\quad + (e_i + e_j)\, m\delta + 2e_i e_j\varepsilon
\end{aligned} \tag{3.9}$$

Note that all the α_{ilj}'s have canceled out. Again, no further progress can be made without additional information about the values of e_i and e_j. Again, assume $e_i \geq e_j$. It follows that Γ_{ij} should have its maximum value, $\left(\underline{\gamma_i} + \delta\right)(e_i + e_j - n - m)$. It also follows that $\overline{\gamma_i}$ should have its maximum value, $\underline{\gamma_i} + \delta$. Substituting into Equation (3.9), and moving e_i and e_j back to the right-hand side gives:

$$\tau > \frac{1}{e_j} m\delta + \frac{1}{e_i} (n + m - e_j)\,\delta + 2\varepsilon$$

Again, both e_i and e_j divide positive quantities, so the right-hand side is maximized when both are equal to ζ. Substituting and solving for ζ gives:

$$\zeta > \delta\,(n + 2m) \,/\, (\delta + \tau - 2\varepsilon) \tag{3.10}$$

The case where $e_i \leq e_j$ proceeds similarly. Minima of $\underline{\gamma_i}\,(e_i + e_j - n - m)$ and $\underline{\gamma_i}$ are substituted for Γ_{ij} and $\overline{\gamma_i}$. Again, it is found that the right-hand side is at a maximum when $e_i = e_j = \zeta$. The result is identical to Equation (3.10).

Equation (3.10) has the same form as Equation (3.7), only $2m$ is used in place of $3m$. As one might expect, restricting the range of the estimates has reduced the number of accepted estimates required for the computation of the synchronization adjustment to be successful. However, the reduction is not large, in the neighborhood of $m/2$. Therefore, restricted range mean is advantageous primarily in instances where m is large, or ζ is near n.

3.3.4 Examples

A few simple examples will help show the utility of these adjustment algorithms.

Start with a 64-node system, and let $\delta = 5$msec., $\tau = 4$msec., and $\varepsilon = 1$msec. When $m = 2$, unrestricted range mean requires 50 accepted estimates at each node, and restricted range mean requires 49. When $m = 5$, unrestricted range mean requires 57 accepted estimates at each node, and restricted range mean requires 53. Unrestricted range mean requires more than 64 accepted estimates (and thus can no longer guarantee synchronization) when $m > 8$, while restricted range mean does not require more than 64 estimates until $m > 12$.

Double the values of δ and τ to 10msec. and 8msec., respectively. Now when $m = 2$, unrestricted range mean requires only 44 accepted estimates at each node, while restricted range mean requires only 43. When $m = 5$, unrestricted range mean requires only 50 accepted estimates, and restricted range mean requires 47. Unrestricted range mean cannot guarantee synchronization for $m > 12$, and restricted range mean cannot guarantee synchronization for $m > 19$. Increasing δ and τ with respect to ε has greatly increased the fault-tolerance.

For a larger example, consider a 1024-node system, and let δ = 5msec., τ = 4msec., and ε = 1msec. When $m = 50$, approximately a 5% failure rate, unrestricted range mean requires 839 accepted estimates at each node, and restricted range mean requires only 803. Unrestricted range mean can guarantee synchronization only when $m \leq 136$, while restricted range mean can guarantee synchronization only when $m \leq 204$. Doubling δ and τ raises these values to $m \leq 204$ and $m \leq 307$, respectively.

3.4 Clock Adjustment With Unreliable Estimates

In the previous section it was assumed that the all estimates were guaranteed to be within their specified maximum uncertainties of the actual skew value. Some types of faults may make it difficult for the estimation algorithm to meet this requirement. Little can be done if there is no relationship between the actual and estimated skew values, but the restrictions can be relaxed a bit.

The calculations of Section 3.3 can be salvaged if the extent of the unreliability can be quantified. Unreliable estimates are modeled as reliable estimates to which have been added an *offset*. The offset is defined as the difference between the computed estimate, and the value the estimate would have had if the fault which is the source of the unreliability were not present. The maximum possible difference between the unreliable estimate and the actual skew value is the sum of its uncertainty and its offset. Given some information about the number of unreliable estimates, and the size of the offsets, the procedures of Section 3.3 can be easily modified to account for this new, enlarged uncertainty.

3.4.1 Constant Offsets

The simplest case is when all offsets have the same value. This is a likely situation with probabilistic estimation algorithms. Most probabilistic estimation algorithms make assumptions about message delays, and a faulty node which invalidates these assumptions will often affect all estimates in the same way.

Let each offset be o, and let N_i and N_j have a_i and a_j unreliable estimates. Consider unrestricted range mean first, re-writing Equation (3.2) to include the offsets gives:

$$\tau \;>\; \alpha_{ji} + \frac{1}{e_j}\left(\Gamma - ij - (e_i + e_j - n - m)\,\alpha_{ji} + \Gamma_j + m\delta + e_j\varepsilon + a_j o\right)$$

$$-\frac{1}{e_i}\left(\Gamma - ij + \Gamma_i - m\delta - e_i\varepsilon + a_i o\right)$$

The derivation continues as it did in Section 3.3.2, until it reaches the point of Equation (3.6):

$$\begin{aligned} e_i e_j \tau \;>\; & (e_i - e_j)\,\Gamma - ij - e_i\,(e_i - n)\,\underline{\gamma_i} + e_j\,(e_j - n)\,\underline{\gamma_i} + \underline{\gamma_i} m\alpha_{ji} \\ & + (e_i + e_j)\,m\delta - e_i\,(e_i - n)\,\delta + 2e_i e_j \varepsilon + e_i a_j o - e_j a_i o \end{aligned}$$

At this point, it is assumed either $e_i \geq e_j$, or $e_j \geq e_i$. Since $e_i \geq a_i$ and $e_j \geq a_j$, it follows that when $e_i \geq e_j$ the maximum value of a_i is greater than or equal to a_j, and the maximum value of the offset terms therefore occurs when $o > 0$ and $a_i = 0$. Similarly, when $e_j \geq e_i$ the maximum value of a_j is greater than or equal to a_i, and the maximum vale of the offset terms therefore occurs when $o < 0$ and $a_j = 0$. The offset term is then either $e_i a_j |o|$, or $e_j a_i |o|$.

As the derivation continues, the offset term becomes either $a_j|o|/e_j$, or $a_i|o|/e_i$. Both terms are maximized when e_i and e_j are ζ. Let a be the maximum number of unreliable estimates at any node. Clearly, the *actual* number of unreliable estimates at any node is no more than ζ. In fact, the actual number of unreliable estimates no more than $\zeta - m$, since the faulty nodes are already assumed to have the maximum possible skews. The final equation for ζ becomes:

$$\begin{aligned} \zeta \;&>\; \tfrac{\delta(n+3m)+a|o|}{\delta+\tau-2\varepsilon}, \quad \text{if } a \leq \zeta - m \\ \zeta \;&>\; \tfrac{\delta(n+3m)-m|o|}{\delta+\tau-2\varepsilon-|o|}, \quad \text{if } a > \zeta - m \end{aligned} \tag{3.11}$$

A similar derivation can be done for range restricted mean. However, in this case the actual number of unreliable estimates at each node is no more than ζ, since adding the same offset to all estimates doesn't affect the range. The result is the following:

$$\begin{aligned} \zeta \;&>\; \tfrac{\delta(n+2m)+a|o|}{\delta+\tau-2\varepsilon}, \quad \text{if } a \leq \zeta \\ \zeta \;&>\; \tfrac{\delta(n+2m)}{\delta+\tau-2\varepsilon-|o|}, \quad \text{if } a > \zeta \end{aligned} \tag{3.12}$$

3.4.2 Bounded Offsets

While constant offsets are perhaps more typical of the unreliable estimates that will be seen in operation, they do not represent all possible faults. More generally, one can bound the value of the offsets, and the number of unreliable estimates at each node.

Assume the absolute value of each offset is no larger than o, and there are no more than a unreliable estimates at each node. The procedure for selecting estimates described in Section 3.2 is then modified to incorporate the offsets into the estimation error. Specifically, each node will choose the largest subset, E, of the estimates such that the average uncertainty of the estimates in E is less than or equal to $\varepsilon - ao/|E|$. No modification of Equation (3.7) or Equation (3.10) is necessary since the offsets are included in the ε terms.

Bounding the offsets instead of assuming constant offsets has made adjustment noticeably more difficult. One obvious requirement is that $\varepsilon > ao/\zeta$. In fact, for practical purposes, ε must be considerably greater than ao/ζ. This implies that either a is much smaller than ζ, or o is much smaller than ε.

3.5 Summary

The interactive convergence algorithm [22] is commonly used to compute synchronization adjustments in distributed systems. However, it was originally designed for completely-connected networks, and assumes that not only will estimates be available for all non-faulty nodes, but that all estimates have the same uncertainty, and that estimates of non-faulty nodes will be reliable. These assumptions may not hold or may be difficult to satisfy in a network that is not completely-connected.

This chapter presented several algorithms which modify the interactive convergence algorithm and relax these restrictions. The principal features of these algorithms are:

- A range of estimate uncertainty is not only allowed, but taken advantage of. Low uncertainty estimates are used to balance high uncertainty estimates to form an average uncertainty.

- Estimates of all non-faulty nodes are not required. This allows estimates with large uncertainties, that would otherwise adversely affect the computation of the synchronization adjustment, to be discarded. Also, transient faults or other circumstances that prevent estimation of otherwise non-faulty nodes can be tolerated.

- The number of faulty nodes is a variable system parameter and is not set at $n/3$.

- The assumption of reliable estimates (estimates of non-faulty nodes are correct), can also be relaxed to some degree.

CHAPTER 4

SYNCHRONIZATION BY GROUPS

The adjustment algorithm of Chapter 3 is a considerable improvement over regular interactive convergence. It does not require nodes spend time and effort attempting to improve estimates of distant nodes. Either better estimates can be used to make up for their shortcomings, or they can be dropped entirely. However, as system sizes increase, even these improvements may not be enough. The complexity of clock distribution algorithm increases at least linearly with system size, in some cases it increases with the square of the system size. The uncertainty of estimates usually increases at least linearly with system diameter. Probabilistic estimation can counteract this effect, but only by causing a considerable increase in the amount of clock information which must be distributed. Finally, the number of estimates each node must make and keep track of increases linearly with system size.

Master/slave adjustment algorithms are one possible alternative. The number of master nodes need not scale linearly with system size, or a hierarchy of masters can be used as in NTP [31, 32]. However, an increase in system size does not eliminate the difficulties of master/slave algorithms, and may in fact exacerbate them. More masters are needed, and with more masters the probability of a master failure increases. The problems of synchronization amongst masters are also increased. If the number of masters does not increase as fast as system size, the average distance to the masters will increase, either increasing estimate uncertainty, or increasing the amount of clock information that must be distributed. Also, the number of nodes which must be served by each master increases, causing more congestion near the master. In hierarchical systems, the depth of the hierarchy increases, and the maximum skew between nodes at the bottom of the hierarchy increases, unless the skew between levels of the hierarchy is reduced by reducing the uncertainty of estimates. Again, any decrease in estimate uncertainty generally requires an increase in the

amount of clock information distributed.

Peer synchronization algorithms would not suffer so much from an increase in system size if nodes did not try to estimate their skew with respect to every other node. Oddly enough, several hardware synchronization algorithms have taken this step. In [39] the number of "estimates" made by each node is reduced by a constant factor. In [33] there is no fixed reduction in estimates, but the fewer the estimates, the slower the system is to stabilize, and the more difficult it is to find a stable configuration. [37] describes a peer synchronization algorithm which uses a network clock distribution algorithm. Nodes in a wrapped square mesh make estimates of only their neighbors. However, the system is only simulated. No analysis is done, and fault-tolerance is not considered.

This chapter introduces an adjustment algorithm which restricts the set of nodes each node must estimate. The number of estimates made by each node is only a fraction of the number of nodes in the system, yet synchronization is preserved. While the maximum skew between any two nodes is likely to be higher than if estimates of all nodes were used, there will still be sets of nodes within which a very small maximum skew will be guaranteed. Cooperating tasks which need tight synchronization may be assigned to nodes in these groups, providing them with the tight synchronization they need, while not requiring the rest of the system pay the high price of such tight synchronization. Fault-tolerance is considered as well, and while reduced, is still reasonably good. And the amount of fault-tolerance can be adjusted by increasing or decreasing the number of nodes each node estimates.

The chapter starts with a brief overview of the algorithm. A graphical approach for analyzing the algorithm is then introduced, and an algorithm for computing the maximum skew in the system is presented. Finally, fault-tolerance is considered, and an algorithm is presented to determine the fault-tolerance of the synchronization algorithm.

4.1 Synchronization Groups

Chapter 3 shows that it is not necessary for each node to make estimates of every other non-faulty node, as long as each node has some minimum number of estimates. The estimate of *any* other node may be discarded, and any number of nodes may discard their estimates of any particular non-faulty node, i.e., all nodes may discard their estimates of N_i, even though N_i is non-faulty. Such a "shunned" node will remain in synchronization because ζ is large enough to ensure that any two nodes will make estimates of many of the same nodes.

It stands to reason that if the set of nodes *not* estimated by any particular node is carefully chosen, that the number of estimates each node must have can be reduced. Seen from the opposite perspective, if one carefully defines which nodes each node estimates, one can make sure that the skew between nodes remains bounded.

A set of *synchronization groups*, $\mathcal{G}$, is defined. Every node is a member of at least one synchronization group, each synchronization group contains at least two nodes, and each synchronization group must intersect at least one other synchronization group. A node estimates only those nodes with which it shares a synchronization group, and the average uncertainty of the estimates must be less than or equal to ε. Estimates outside the range $[-\hat{\delta}, \hat{\delta}]$ are discarded, and the remaining estimates are averaged to get the synchronization adjustment. The synchronization groups must be defined so that the maximum skew between any two nodes in the same synchronization group is $\hat{\delta}$, nodes which belong to multiple synchronization groups provide the means of bounding the skew between nodes that do not belong to a common synchronization group. For example, if N_i and N_j do not belong to a common synchronization group, but there exists N_h and synchronization groups G_1 and G_2 such that $N_i, N_h \in G_1$ and $N_j, N_h \in G_2$, then the maximum skew between N_i and N_j is $2\hat{\delta}$. The value of $\hat{\delta}$ is chosen small enough that the maximum skew between any two nodes is δ.

What the above arrangement means is that the tightness of synchronization between nodes varies. Nodes that belong to the same synchronization group will have a maximum skew of $\hat{\delta}$, while nodes that do not belong to a common synchronization group may have much larger skews. This implies that tasks which need to be tightly synchronized with one another should be assigned to nodes in the same synchronization group. These tasks get tight synchronization, but the whole system does not have to pay the cost of that synchronization just for their benefit.

It is non-trivial to demonstrate that a specific set of synchronization groups has the property that the skew between any two members of a synchronization group is no greater than δ. Much of the remainder of this chapter will be devoted to this task. The following definitions are the first step in this process:

Definition 4.1 *Two nodes, N_i and N_j, are said to be* tied *if there exists a synchronization group, G, such that $N_i \in G$ and $N_j \in G$. The relationship is symmetric, if N_i is tied to N_j, then N_j is tied to N_i.*

Definition 4.2 *The* synchronization set *of N_i is the set of nodes with which it is tied.*

Definition 4.3 *The transitive closure of tied is* strung. *Two nodes, N_i and N_j, are strung if and only if there exists a sequence of nodes, $N_{a_1}, \ldots, N_{a_b}$, such that N_i is tied to N_{a_1}, N_j is tied to N_{a_b}, and N_{a_h} is tied to $N_{a_{h+1}}$ for all $1 \leq h < b$.*

A node makes estimates of only those nodes with which it is tied, i.e., those nodes in its synchronization set. While a node need not be tied to every other node, it must be strung to every other node. Only if two nodes are strung may there exist a bound on the skew between them.

As a simple example, consider a sixteen-node hypercube. Each node has a binary address between 0000 and 1111. Define eight synchronization groups of four members each. The first four groups are the subcubes of the form $ab**$, and the second four groups are the subcubes of the form $**ab$, where $*$ indicates a "don't care" address bit and $a, b \in \{0, 1\}$. Every node belongs to two synchronization groups, and there are three other nodes in each group. Each node has six nodes in its synchronization set, e.g., the synchronization set of node 0000 is $\{0001, 0010, 0011, 0100, 1000, 1100\}$, and any pair of nodes is strung.

4.2 The Synchronization Graph

A bipartite *synchronization graph* can be derived from the definitions of the synchronization groups. The two vertex sets are as follows:

Group vertices: For every $G \in \mathcal{G}$, there exists a group vertex, g_G.

System vertices: For every N_i, there exists a system vertex, s_i.

An edge exists between s_i and g_G if and only if $N_i \in G$. Therefore, N_i and N_j are tied if and only if the distance between s_i and s_j is 2. And, N_i and N_j are strung if and only if there is a path between s_i and s_j. Finally, all pairs of nodes will be strung if and only if the synchronization graph is connected.

Figure 4.1 shows the synchronization graph for the sixteen-node hypercube example discussed in the previous section, where the small circles represent system vertices and large circles are group vertices.

Definition 4.4 *The* stretch *between N_i and N_j is* half *the length of the shortest path between s_i and s_j.*

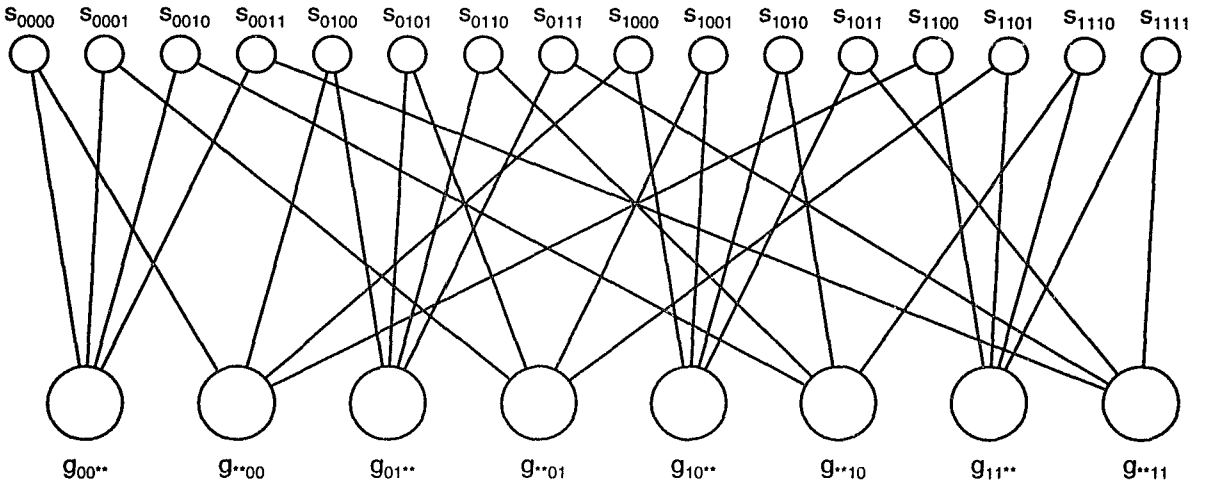

Figure 4.1: Synchronization graph for a 16-node hypercube multicomputer

The stretch between two tied nodes is 1. If the stretch between N_i and N_j is h, then the maximum skew between N_i and N_j is $h\hat{\delta}$. The maximum stretch between any two nodes determines the relative values of $\hat{\delta}$ and δ, if the maximum stretch is h, then $\hat{\delta} \leq \delta/h$. In Figure 4.1, the distance between any two system vertices is no greater than 4, so the stretch between any two nodes is no greater than 2, and $\hat{\delta} \leq \delta/2$. One should take care to distinguish between the stretch between two nodes, and the distance between them in the multicomputer's network. The stretch indicates the maximum skew between two nodes, and is not necessarily related to the physical distance between nodes. This can be seen in Figure 4.1, where the maximum stretch is 2, but the diameter of a 16-node hypercube is 4.

4.2.1 Synchronization Paths

A pair of tied nodes will make estimates of some of the same nodes, but each will likely make estimates of some nodes that the others do not. For example, in the hypercube 0000 and 0001 will both make estimates of 0010 and 0011, but of these two nodes only 0000 will make an estimate of 1000, and only 0001 will make an estimate of 1001. At first glance it may seem that 1000 and 1001 could have a skew of as much as $3\hat{\delta}$, which could make it difficult for 0000 and 0001 to stay within $\hat{\delta}$ of not only each other, but of the rest of their synchronization sets as well. Closer inspection reveals that 1000 and 1001 are tied, and the maximum skew between them is therefore $\hat{\delta}$. This makes it much easier for 0000 and 0001 to stay within $\hat{\delta}$ of their respective synchronization sets. Analysis of these types of relationships is important when determining if a particular set of synchronization groups allows the system to remain synchronized.

The following notation will be useful in the discussion that follows:

S_i: The set of all system vertices a distance of 2 from s_i. These are the system vertices which correspond to nodes in N_i's synchronization set.

$S_{ij}^{\cap}$: The intersection of S_i and S_j. These are the system vertices corresponding to nodes in *both* N_i's and N_j's synchronization sets.

G_i: The set of group vertices a distance of 1 from s_i. These are the group vertices corresponding to the synchronization groups which contain N_i.

$G_{ij}^{\cap}$: The set of group vertices a distance of 1 from *both* s_i and s_j. These are the group vertices corresponding to synchronization groups containing both N_i and N_j.

Definition 4.5 *Given synchronization graph S, and tied nodes N_i and N_j, a* synchronization path *(SP) is a simple (non-self-intersecting) path in S from a member of S_i to a member of S_j, which contains at most one member of $S_{ij}^{\cap}$, no members of $G_{ij}^{\cap}$, and has length no greater than* 4.

The synchronization paths of N_i and N_j are used to determine the maximum skew between N_i and N_j immediately after synchronization. The existence of an SP between $s_{i'} \in S_i$ and $s_{j'} \in S_j$ indicates a relationship between $N_{i'}$ $N_{j'}$. Either $N_{i'}$ and $N_{j'}$ are tied to each other, or are both tied to some common third node. In either case the skew between $N_{i'}$ and $N_{j'}$ is limited to either $\hat{\delta}$or $2\hat{\delta}$. This in turn limits the maximum possible skew between N_i and N_j immediately after synchronization.

As an example, consider Figure 4.1 again. 0000 and 0001 are both in synchronization group $00**$. The following sets are defined:

$$\begin{aligned} S_{0000} &= \{s_{0001}, s_{0010}, s_{0011}, s_{0100}, s_{1000}, s_{1100}\} \\ S_{0001} &= \{s_{0000}, s_{0010}, s_{0011}, s_{0101}, s_{1001}, s_{1101}\} \\ G_{0000\ 0001}^{\cap} &= \{g_{**00}, g_{**01}\}. \end{aligned}$$

Each of the members of S_{0000} not in $S_{0000\ 0001}^{\cap}$ has an SP of length 2: $s_{0100} \rightarrow g_{01**} \rightarrow s_{0101}$, $s_{1000} \rightarrow g_{10**} \rightarrow s_{1001}$, and $s_{1100} \rightarrow g_{11**} \rightarrow s_{1101}$. There are also 12 SPs of length 4. The SPs show that the maximum skew between members of 0000's synchronization set and 0001's synchronization set is $\hat{\delta}$, significantly less than the $3\hat{\delta}$ that was first supposed.

```
Procedure find_synch_path(fragment)
begin
    if fragment is a synchronization path then
        makeSP(fragment)
    else
        if length(fragment) < 4 then
            tail = last vertex of fragment;
            foreach neighbor of tail
                find_synch_path(fragment+neighbor);
        endif
    endif
end
```

Figure 4.2: Procedure *find_synch_path*

Each SP has a corresponding stretch of half the length of the SP. SPs are classified according to their corresponding stretches. An SP of length 2 has a corresponding stretch of 1, and is therefore called a *1-SP*. Similarly, an SP of length 4 is called a *2-SP*.

All SPs for a given pair of tied nodes can be found using a simple modification of a depth-first search algorithm. Procedure *find_synch_path*, shown in Figure 4.2, finds all SPs which have endpoints at a given vertex. The complexity of this algorithm depends on the synchronization graph. If each node belongs to no more than g synchronization groups, and each synchronization group has no more than k members, then the maximum size of S_i is $g(k-1)$. The algorithm will search all paths of length 4 from these nodes, for a maximum of $g(k-1)g^2k^2 = g^3k^2(k-1)$ paths.

4.2.2 Synchronization Areas

Showing that the system remains synchronized is done in a manner similar to that of Chapter 3. It is assumed that when the adjustment algorithm starts, the maximum skew between nodes in the same synchronization group is $\hat{\delta}$. Then it is shown that after the synchronization adjustments have been added to the target adjustments, the maximum skew between nodes in the same synchronization group is less than some $\hat{\tau}$, $\hat{\tau} \leq \hat{\delta}$. This could require checking every pair of tied nodes to make sure this condition holds. However,

usually the number which need to be checked is much lower.

Definition 4.6 *Given synchronization graph S, and tied nodes N_i and N_j, the* synchronization area *of N_i and N_j is a subgraph of S containing s_i, s_j, G_i, G_j, S_i, S_j, all the edges which connect these vertices, and all vertices and edges contained in SPs for N_i and N_j.*

The synchronization area of N_i and N_j is all vertices and edges contained in paths of length less than or equal to 8 between s_i and s_j. This will contain all SP's for vertices in S_i and S_j. Since the SP's determine the maximum skew between N_i and N_j, the synchronization area is the only part of the synchronization graph that has any part in computing the maximum skew. One should take care to distinguish between a *synchronization set* and a *synchronization area.* A synchronization set is the set of nodes whose skews a node estimates in order to synchronize. A synchronization area is the subgraph of the synchronization graph which is used to determine the maximum skew for a pair of tied nodes.

Any pair of tied nodes will have a corresponding synchronization area. If two synchronization areas differ only in the labeling of their vertices, i.e., one can be transformed to the other by simply relabeling its vertices, then they are said to be *equivalent.* To show the system is synchronized, one has to show synchronization for all possible non-equivalent synchronization areas, i.e., any synchronization areas equivalent to synchronization areas already checked don't have to be checked. This greatly reduces the number of cases. Often, as in Figure 4.1, the synchronization graph will be identical from the point of view of any system vertex. More specifically, given N_i, a labeling of the vertices of the synchronization graph can be found which gives the label s_i to any desired system vertex. In Figure 4.1, the label s_{0000} is given to the system vertex at the far left; it could just have easily been given to the system vertex at the far right, or any system vertex in between. In such cases, all synchronization areas are equivalent.

4.3 Calculation of Maximum Skew

The proof of synchronization was outlined briefly in Section 4.2.2, and is similar to the proofs used in Chapter 3. One starts by assuming that the maximum skew between any two tied nodes is less than $\hat{\delta}$. Then one shows that immediately after all nodes have adjusted their clocks, the maximum skew between tied nodes is less than $\hat{\tau}$, $\hat{\tau} \leq \hat{\delta}$. Chapter 3 has already done similar proofs in the case where there is a single synchronization group containing all nodes. In this case, things are more complicated, and in this section it is

shown how to prove the system will remain synchronized for a given set of synchronization groups, $\mathcal{G}$. It should be understood that this only shows how to *prove* the algorithm works, it does not describe the algorithm's operation. None of the computations done in this section have to be made by the synchronization algorithm during operation.

Assume N_i and N_j are tied. Let e_i and e_j be the number of skew estimates made by N_i and N_j. Let Λ_i and Λ_j be the *sums* of the skew estimates computed by N_i and N_j. Assume, without loss of generality, that N_j has a greater clock value than N_i. At worst, the skew between N_i and N_j is already the maximum allowable, $\hat{\delta}$. The maximum skew between N_i and N_j after synchronization can then be found by maximizing the following quantity:

$$\hat{\delta} + \frac{\Lambda_j}{e_j} - \frac{\Lambda_j}{e_i} \tag{4.1}$$

In order to show synchronization, it must be less than $\hat{\tau}$, i.e.

$$\hat{\tau} \geq \hat{\delta} + \left(\frac{\Lambda_j}{e_j} - \frac{\Lambda_i}{e_i} \right) \tag{4.2}$$

Consider the synchronization area of N_i and N_j. Assume that the skew of N_j with respect to N_i is $\hat{\delta}$. Each vertex in S_i and S_j should be given a skew, with respect to the appropriate node, so that if these were the skews computed for their respective nodes the value in Equation (4.1) would be maximized. At worst, members of S_j are given skews of $\hat{\delta}$ with respect to N_j, and members of S_i are given skews of $-\hat{\delta}$ with respect to N_i. This implies a skew of $3\hat{\delta}$ between members of S_j and S_i. But, some vertices will be in both S_i and S_j, and a vertex must have a skew of 0 with respect to itself. Also, an SP imposes a limit on the skew between its endpoints. A 1-SP indicates a maximum skew of $\hat{\delta}$ between its endpoints, and a 2-SP indicates a maximum skew of $2\hat{\delta}$ between its endpoints. By giving a skew to one vertex one limits the skews which can be given to a number of other vertices, and by giving skews to these vertices one limit the skews which amy be given to even more vertices, and so on. Finding the maximum skew between N_i and N_j after synchronization is therefore a process of searching a large number of cases.

Searching the individual cases is much too slow, there must be a way of limiting the number of individual cases. To do this, the problem is broken into a number of similar, but smaller ones. The division is done carefully, so that the maximum for each of the small problems can be found by checking only a few cases. The sum of these maxima is an upper bound for the true maximum, and in many cases will be equal to it.

4.3.1 Clusters

The problem is one of maximizing a quantity subject to certain restrictions. Removing restrictions will not reduce the maximum, so it is safe to ignore some restrictions, since the result will be an increase in the calculated maximum skew. The SPs correspond to the restrictions, and the problem is simplified by eliminating many of the SPs. The members of S_i and S_j are partitioned into *clusters.* A cluster is a group of vertices where each vertex has an SP to at least one other vertex in the cluster. Any SPs between clusters (and many SPs within a cluster) are ignored, allowing the consideration of each cluster separately. Finding the maximum skew for each cluster is a simple matter of checking a few cases. The sum of the maximums for each cluster is then an upper bound for the actual maximum skew.

A vertex in either S_i or S_j can be typed by the length of the shortest SP for which it is an endpoint: a vertex in $S_{ij}^{\cap}$ is an *intersection* vertex, a vertex which is the endpoint of a 1-SP is a *1-vertex*, a vertex which is the endpoint of a 2-SP (but no 1-SPs) is a *2-vertex*, and a vertex which is not the endpoint of any SP is an *unbound* vertex. To form clusters, each 1-vertex and 2-vertex will be *assigned* to some other vertex. An assignment indicates the existence of an SP, and thus a bound on the skew, between two vertices. If vertex s_a is to be assigned to vertex s_b, the following two requirements must be met:

1. There must be an SP with endpoints at s_a and s_b.

2. If s_a is a 1-vertex, there must be a 1-SP with endpoints at s_a and s_b. Note that s_b then must be either a 1-vertex or an intersection vertex.

Because each assignment corresponds to some SP, either $s_a \in S_i$ and $s_b \in S_j$, or the reverse. Also, notice that a 1-vertex must be assigned to either a 1-vertex or an intersection vertex, while a 2-vertex can be assigned to either a 2-vertex, a 1-vertex, or an intersection vertex.

A cluster is a minimal non-empty set of vertices such that for every vertex s_a in the cluster, the cluster will contain all vertices assigned to s_a, and the vertex to which s_a is assigned, if any (an intersection vertex may belong to a cluster if some vertex is assigned to it, but it will not be assigned to any vertex). As an example, if s_a is assigned to s_b, and some vertex s_c is assigned to s_a, all three vertices will be in the same cluster. Because a cluster is a minimal set, no subset can be removed and still leave a cluster.

No special effort needs to be made to find clusters, they can be found as a direct result of making assignments. To find the clusters, make assignments one at a time according to

the following procedure:

1. Assign 1-vertices first, then 2-vertices.
2. If s_a is assigned to s_b and s_b already belongs to a cluster, then s_a belongs to s_b's cluster.
3. If s_a is assigned to s_b and s_b does not belong to any cluster, a new cluster is created with s_a and s_b as members. Furthermore, if s_a and s_b are the same type vertices (i.e., both 1-vertices or both 2-vertices) assign s_b to s_a. Notice that if s_B is a 1-vertex, then s_a must be a 1-vertex because all 1-vertices are assigned before 2-vertices.

A cluster where all vertices are assigned to vertices of the same type is called a *straight* cluster. If one or more vertices is assigned to a vertex of a different type (e.g., a 1-vertex is assigned to an intersection vertex), the cluster is called *jumbled.* Calculation of maximum skew is easiest when clusters are small and straight. To get small, straight clusters one must be careful when selecting assignments. If vertex s_a is to be assigned, then for each s_b to which s_a could be assigned, place s_b in whichever of the following sets is appropriate:

1. Vertices of the same type as s_a which do not belong to a cluster.
2. Vertices of the same type as s_a which belong to a straight cluster.
3. Intersection vertices which do not belong to a cluster.
4. Vertices which belong to a jumbled cluster.
5. Vertices of a different type than s_a which belong to a straight cluster.

These sets are listed in order of decreasing desirability. The vertex to which s_a is assigned is selected from the most desirable non-empty set. Within sets 1 and 3, select one at random. Within sets 2, 4, and 5, select at random from the vertices which belong to the smallest clusters.

4.3.2 Computing Skew Terms

Clusters reduce the maximizing problem of Equation (4.2) to one of maximizing a sum of terms. The terms are formed by breaking up the Λ_i and Λ_j sums and reorganizing and mixing pieces to form terms of the form $x/e_j - y/e_i$. The form of x and y is a sum of

"related" skews. For example, the term for a cluster will have x as the sum of skews for vertices in the cluster which are in S_j, and y as the sum of skews for vertices in the cluster which are in S_i. There will be one term for each cluster, one term for estimation error, one term for the skew between N_i and N_j, one term for the unbound vertices, and one term for the intersection vertices which do not belong to a cluster. The maximum of the sum is found by maximizing each term, which means maximizing the values of the skews. Because of dependencies between terms (due to SPs between clusters which are being ignored), the maximum of the sum will be an upper bound on the actual maximum skew between N_i and N_j after synchronization.

The maximum of each term is the maximum "contribution" each term may make to the skew between N_i and N_j after synchronization. In most cases, finding the maximum means checking several possible worst-case configurations of vertices to see which is the maximum. The rest of this section lists these worst case configurations, and shows how to compute the skew for each. The details are tedious, and the casual reader may wish to proceed directly to Section 4.3.3.

Non-Cluster Terms

The error term represents the maximum contribution to the skew due to estimation uncertainty. If the maximum uncertainty is ε, estimation uncertainty can subtract $\varepsilon\,(e_i - 1)$ from Λ_i and add $\varepsilon\,(e_j - 1)$ to Λ_j (there is no error in estimating one's own clock). The value of this term is then $\varepsilon\frac{e_j-1}{e_j} + \varepsilon\frac{e_i-1}{e_i}$.

The next term is generated by each node estimating the other's clock. If the skew between them is $\hat{\delta}$, and N_j has the greater clock value, the value of this term is $-\hat{\delta}/e_j - \hat{\delta}/e_i$. This term will always reduce the maximum skew.

Because the unbound vertices have no SPs, their skew values are bound only because they belong to either S_i or S_j. Therefore, as a worst case, the unbound vertices in S_i are given skews $-\hat{\delta}$, and the unbound vertices in S_j are given skews $\hat{\delta}$. If there are $e_{i,u}$ unbound vertices in S_i and $e_{j,u}$ unbound vertices in S_j, this term has value $\hat{\delta}\left(\frac{e_{j,u}}{e_j} + \frac{e_{i,u}}{e_i}\right)$.

Intersection vertices are given skews relative to both N_i and N_j. These skews must be consistent, i.e., for a given vertex the skew with respect to N_i must be $\hat{\delta}$ greater than the skew with respect to N_j. If there are $e_{\cap,u}$ intersection vertices which do not belong to any cluster, and the total skew with respect to N_i of these vertices is $\Lambda_{\cap,u}$, then the value of this term is $\left(\Lambda_{\cap,u} - \hat{\delta}e_{\cap,u}\right)/e_j - \Lambda_{\cap,u}/e_i$. This value is not constant, but is a function of $\Lambda_{\cap,u}$.

Because the intersection vertices must remain within $\hat{\delta}$ of *both* N_i and N_j, their skews with respect to N_i must be in $[0, \hat{\delta}]$. This gives $\Lambda_{\cap,u}$ a range of $[0, \hat{\delta}e_{\cap,u}]$. The function is linear, so it has its maximum at one of the endpoints. The maximum value of this term is then the maximum of $-\hat{\delta}e_{\cap,u}/e_j$ and $-\hat{\delta}e_{\cap,u}/e_i$. This term also will only reduce the maximum skew.

Cluster Terms

Each cluster will generate a term in the sum, and the form of the term depends on the type of cluster. The general idea is to place vertices from S_i as far from the vertices of S_j as the SPs will allow, while keeping their skews less than $\hat{\delta}$. This is similar to the original problem of giving skews to vertices discussed at the beginning of Section 4.3, only now matters have been greatly simplified by considering only one cluster at a time, and by ignoring SPs between clusters. Matters are simplified even further by considering only those SPs within a cluster which correspond to the actual assignments. A *configuration* of a cluster is made by giving a skew to each of its members, and each cluster will have only a few possible configurations of its members which could yield a maximum value for its term.

Number the clusters from 1 to M, where M is the total number of clusters. The following notation is used throughout the rest of this section:

$S^c_{i,1}$: The set of 1-vertices in cluster c which are in S_i, but not in S_j.

$e^c_{i,1}$: The number of vertices in the set $S^c_{i,1}$.

$S^c_{i,2}$: The set of 2-vertices in cluster c which are in S_i, but not it S_j.

$e^c_{i,2}$: The number of vertices in the set $S^c_{i,2}$.

The notation is defined similarly for N_j.

The preference for assigning vertices to unassigned vertices has an interesting consequence for straight clusters of 1-vertices. Every vertex in $S^c_{i,1}$ is assigned to the same vertex in $S^c_{j,1}$ (the *point* of $S^c_{j,1}$), and every vertex in $S^c_{j,1}$ is assigned to the same vertex in $S^c_{i,1}$ (the point of $S^c_{i,1}$). This results in two possible configurations for the maximum skew. A *whole* configuration is one where all the vertices are given the same skew k. In a *fractured* configuration all but the point of $S^c_{i,1}$ and the point of $S^c_{j,1}$ are given skews $-\hat{\delta}$, while all but the point of $S^c_{j,1}$ and the point of $S^c_{i,1}$ are given skews $\hat{\delta}$. Figure 4.3 shows the two configurations, the fractured configuration on top and the whole configuration on the bottom.

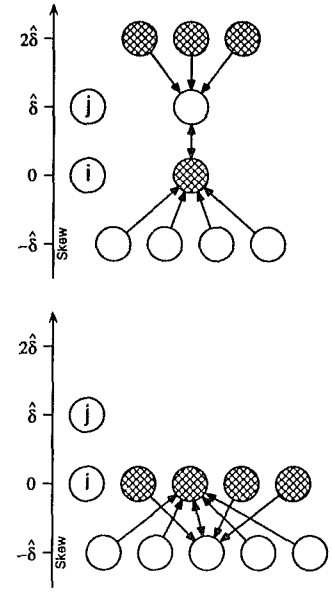

Figure 4.3: Configurations for a straight cluster of 1-vertices

The line at left shows skew with respect to N_i, subtracting $\hat{\delta}$ from this value gives skew with respect to N_j. The vertices of $S^c_{j,1}$ are cross-hatched, and assignments are indicated by the arrows. The whole configuration always yields the maximum value when either $S^c_{i,1}$ or $S^c_{j,1}$ has only one member. This configuration generates a term similar to the one for intersection vertices, i.e., it is a linear function of the skew given to the vertices. The function will have its maximum when the skews are either $-\hat{\delta}$ or $\hat{\delta}$. The fractured configuration usually yields the maximum when both $S^c_{i,1}$ and $S^c_{j,1}$ have more than one member. There is only one possible maximum for the fractured configuration. Equation (4.3) shows the skew term for a straight cluster of 1-vertices. The top two equations are generated by the whole configuration, the bottom equation is generated by the fractured configuration.

$$\max \begin{cases} \hat{\delta}\left(e^c_{i,1}/e_i - e^c_{j,1}/e_j\right) \\ \hat{\delta}\left(e^c_{j,1}/e_j - e^c_{i,1}/e_i\right) \\ \hat{\delta}\left(\left(e^c_{j,1}-2\right)/e_j + \left(e^c_{i,1}-2\right)/e_i\right) \end{cases} \tag{4.3}$$

A straight cluster of 2-vertices is handled much like the straight cluster of 1-vertices. The same general configurations apply, and only the values of the skews change slightly since nodes may now be $2\hat{\delta}$ apart. Equation (4.4) shows the skew term for a straight cluster of 2-vertices. Again, the top two equations are generated by the whole configuration, and the bottom equation is generated by the fractured configuration.

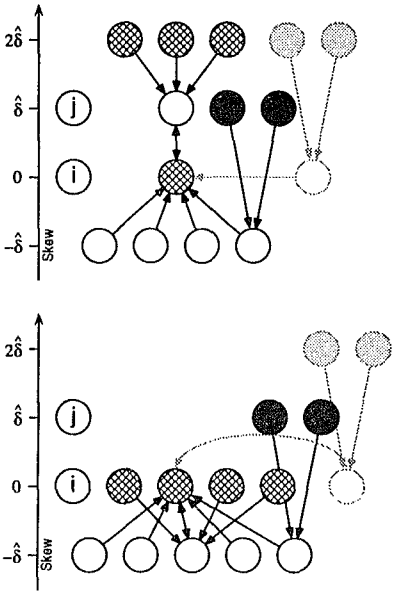

Figure 4.4: Configurations for a jumbled cluster without an intersection vertex

$$\max \begin{cases} \hat{\delta} e^c_{i,2}/e_i \\ \hat{\delta} e^c_{j,2}/e_j \\ \hat{\delta}\left(\left(e^c_{j,2}-1\right)/e_j + \left(e^c_{i,2}-1\right)/e_i\right) \end{cases} \tag{4.4}$$

Jumbled clusters can be one of two kinds: either they contain an intersection vertex or they do not. Each will be considered separately. Jumbled clusters are somewhat more complex than straight clusters, the following definitions are needed:

Definition 4.7 *A* sub-cluster *is formed whenever a 2-vertex is assigned to a 1-vertex. The sub-cluster consists of the 1-vertex and all 2-vertices which have been assigned to it.*

Definition 4.8 *A vertex in a jumbled cluster is* free *if it does not belong to any sub-cluster.*

The existing notation is extended to include free vertices:

$e^c_{i,1f}$: The number of free 1-vertices in $S^c_{i,1}$.

$e^c_{i,2f}$: The number of free 2-vertices in $S^c_{i,2}$.

If a jumbled cluster does not contain an intersection vertex, then it is simply a straight cluster of 1-vertices where some 2-vertices have been assigned to some of the 1-vertices (forming sub-clusters). Whole and fractured configurations exist just as in straight clusters.

The main difference is the extra consideration that must be given to the sub-clusters. In both configurations shown in Figure 4.4 a sub-cluster has been formed by the assignment of pair of 2-vertices (gray filled) to a single 1-vertex. It may be the case that the term will have a greater value if the three vertices had the positions shown in light gray. For this reason, sub-clusters must be considered separately from the rest of the cluster. As an example, Figure 4.4 contains only one sub-cluster, the one discussed above. It contains three vertices, and its value is $\max\{-\hat{\delta}, 2\hat{\delta}\}$ (the two possible terms generated by the sub-cluster).

With some modification for sub-clusters, computation of maximum term value proceeds much like it does for straight clusters. The whole configuration is somewhat more complex as it will now have three possible maxima. The first two are the same as for the straight cluster, the free vertices are given skews $-\hat{\delta}$ or $\hat{\delta}$. When they are given skews $-\hat{\delta}$ (as shown in the lower part of Figure 4.4) sub-clusters containing vertices in $S^c_{i,1}$ must be considered separately, while when they are given skews $\hat{\delta}$ sub-clusters containing vertices in $S^c_{j,1}$ must be considered separately. The third possible maximum is brought about by the presence of 2-vertices. Members of $S^c_{i,1}$ and $S^c_{i,2}$ will have skews 0 and $-\hat{\delta}$, while members of $S^c_{j,1}$ and $S^c_{j,2}$ will have skews 0 and $\hat{\delta}$, and sub-clusters are not considered. The fractured configuration in the upper half of Figure 4.4 is handled just like the analogous arrangement for straight clusters, except the sub-clusters are handled separately. The term is computed as if the sub-cluster vertices did not belong to the cluster, then the sub-cluster terms are added in. Equation (4.5) shows the skew terms for a jumbled cluster which does not contain an intersection vertex. The top two equations are almost the same as those for the whole configuration of straight clusters of 1-vertices, except only free nodes are used, and sub-cluster terms are added. The third equation comes from the third possible maxima of the whole configuration. The final equation is for the fractured configuration.

$$\max \begin{cases} \hat{\delta}\left(e^c_{i,f1}/e_i - e^c_{j,f1}/e_j\right) + \textit{ sub-clusters} \\ \hat{\delta}\left(e^c_{j,f1}/e_j - e^c_{i,f1}/e_i\right) + \textit{ sub-clusters} \\ \hat{\delta}\left(e^c_{j,2}/e_j + e^c_{i,2}/e_i\right) + \textit{ sub-clusters} \\ \hat{\delta}\left(\left(e^c_{j,f1} - 2\right)/e_j + \left(e^c_{i,f1} - 2\right)/e_i\right) + \textit{ sub-clusters} \end{cases} \tag{4.5}$$

If a jumbled cluster contains an intersection vertex, it must contain only one intersection vertex, and every vertex in the cluster must be assigned to a vertex of a different type. It follows that all 1-vertices, and perhaps some of the 2-vertices, is assigned to the sole intersection vertex. The 2-vertices assigned to 1-vertices form sub-clusters, as in the case

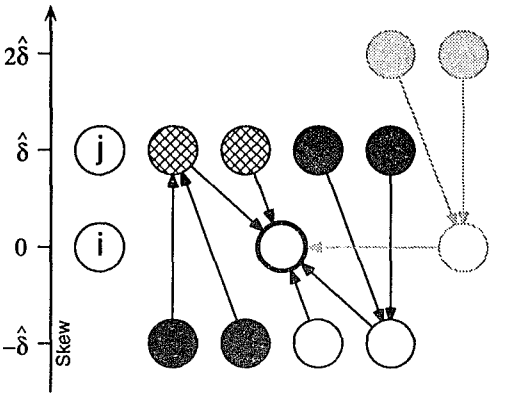

Figure 4.5: Configurations for a jumbled cluster containing an intersection vertex

of jumbled clusters without intersection vertices. Once again the result is a linear function for the value of the term, this time depending on the skew given to the intersection vertex. Thus, there are two possible maxima, when the intersection vertex has a skew of either 0 or $\hat{\delta}$with respect to N_i. The configuration where it is given 0 is shown in Figure 4.5. The intersection vertex is shown with a bold border. In this configuration any sub-clusters containing vertices of $S^c_{i,1}$ must be considered separately. If the intersection vertex is given skew $\hat{\delta}$ with respect to N_i, then any sub-clusters containing vertices of $S^c_{j,1}$ must be considered separately. Equation (4.6) shows the skew terms for a jumbled cluster containing an intersection vertex. The top equation is for the case when the intersection node is given skew 0, and the bottom equation is for the case where the intersection vertex is given skew $\hat{\delta}$.

$$\max \begin{cases} \hat{\delta}\left(\left(e^c_{j,f2}-1\right)/e_j + \left(e^c_{i,f1}+e^c_{i,2}\right)/e_i\right) + \textit{ sub-clusters} \\ \hat{\delta}\left(\left(e^c_{j,f1}+e^c_{j,2}\right)/e_j + \left(e^c_{i,f2}-1\right)/e_i\right) + \textit{ sub-clusters} \end{cases} \tag{4.6}$$

4.3.3 Relating ε, $\hat{\delta}$, and $\hat{\tau}$

If synchronization is to be maintained it must shown that the value in Equation (4.1) is less than or equal to $\hat{\delta}$, i.e.,

$$\hat{\tau} \geq \hat{\delta} + \Lambda_j/e_j - \Lambda_i/e_i \tag{4.7}$$

From the results of Section 4.3.2 it can be seen that Λ_i and Λ_j are functions of ε and $\hat{\delta}$. So Equation (4.7) relates three variables, ε, $\hat{\delta}$, and $\hat{\tau}$. If the value of one of them is known, one can solve to get a minimum ratio for the remaining two, e.g., if ε is 1 msec. one can substitute into Equation (4.7) to get a minimum ratio between $\hat{\delta}$and $\hat{\tau}$. Even if

only the ratio of two of the variables is known, the minimum ratio for the other two can be determined, e.g., if $\hat{\delta} = 10\varepsilon$ substituting into Equation (4.7) will produce a minimum ratio between $\hat{\delta}$ and $\hat{\tau}$.

As an example, consider the 16-node hypercube whose synchronization graph is shown in Figure 4.1. The synchronization set of each node has six members, for a total of seven estimates (including the 0 estimate for its own clock). For any pair of tied nodes, there will be 2 intersection vertices, and 6 1-vertices (3 for each node). The estimation uncertainty will contribute no more than $\frac{12}{7}\varepsilon$ to the maximum skew. Each node estimating the other's clock will contribute no more than $-\frac{2}{7}\hat{\delta}$ to the maximum skew. The two intersection vertices will also contribute no more than $-\frac{2}{7}\hat{\delta}$ to the maximum skew. Cluster assignment yields 3 straight clusters of 1-vertices, each with two members, and each contributing 0 to the maximum skew. The result is the following inequality:

$$\begin{aligned} \tau &\geq \hat{\delta} + \frac{12}{7}\varepsilon - \frac{4}{7}\hat{\delta} \\ &\geq \frac{12}{7}\varepsilon + \frac{3}{7}\hat{\delta} \end{aligned}$$

If $\hat{\delta} = 10\varepsilon$, then $\hat{\tau} \geq \frac{3}{5}\hat{\delta}$, or $\hat{\tau} \geq 6\varepsilon$. A good estimation algorithm might have an ε of 1 msec., then $\hat{\delta}$is 10 msec., and $\hat{\tau}$is at least 6 msec. If $\varrho = 10^{-6}$, Equation (3.1) gives a time between synchronizations of no more than 4000 sec.

4.3.4 Complexity

Assigning vertices to clusters requires looking at each SP at most twice, and thus has a complexity of no more than twice the number of SPs. Computing the value of each cluster requires looking at each vertex once, so the complexity is related to the number of vertices. Thus it is cluster assignment which dominates the complexity of computing maximum skew.

In practice, cluster assignment is very fast. All SPs need not be considered, only the shortest ones. Usually, only a few of these need to be checked in order to make an assignment. Also, in practice one need not check all the vertices of a cluster in order to compute its maximum value. Most of the information (the cluster type, whether or not it contains an intersection vertex, the numbers of nodes of each type) can be derived as the cluster is formed. Only sub-clusters require special handling.

4.3.5 Defining Synchronization Groups

So far discussion has assumed that a set of synchronization groups, $\mathcal{G}$, is defined, and shown how to compute the maximum skew for them. Nothing has been said about how to define them for a particular system. For some systems there is an obvious choice for the synchronization groups. For instance, in the hypercube it is reasonable to base the synchronization groups on subcubes. However, there is no general algorithm that works well for all possible systems. What constitutes a good set of synchronization groups will depend, in part, on the network and the distance between nodes. Clock information is usually more accurate and easier to get for nodes that are close by, so defining synchronization groups that contain many nodes from distant parts of the system can make synchronization more difficult and less accurate.

There is, however, a simple approach which can be used to generate good synchronization groups for most systems. For an n node system, lay out the nodes in a $\sqrt{n} \times \sqrt{n}$ grid, filling any empty spaces with "dummy" nodes. Define each row and each column of the grid to be a synchronization group. This results in a fairly good set of $2\sqrt{n}$ synchronization groups of $\sqrt{n}$ nodes each. If more synchronization groups are needed (for fault-tolerance), one can define the diagonals as synchronization groups as well, yielding $4\sqrt{n}$ synchronization groups of $\sqrt{n}$ nodes each.

This approach may not be ideal under all circumstances, but it may serve as a good starting point. One particular advantage is that the synchronization graph will be fairly homogeneous, so few synchronization areas will have to be checked when computing the maximum skew. Furthermore, if $\sqrt{n}$ is an integer, the synchronization graph will be homogeneous, so there will only be one synchronization area to check.

4.4 Fault-Tolerance

Any multicomputer, especially a large one, or one which will be operating for a long period of time, will have to deal with faults. Such systems are designed to tolerate a given number of faults, and the synchronization algorithm must tolerate these faults too.

Standard adjustment algorithms use estimates of every other node (or the vast majority of other nodes). Using synchronization groups reduces the number of estimates at each node, and reduces the worst-case fault-tolerance as well. Since each node makes fewer estimates, each fault has a greater effect. However, considerable fault-tolerance is retained,

and the reduction in the number of estimates per node reduces the probability that a faulty node will even be seen.

4.4.1 Fault Model

Any analysis of fault-tolerance depends upon the fault model used. Since few assumptions are made about the way in which clock information is distributed, it is assumed that estimates are *reliable*, just as it was in Section 3.3. This may be accomplished by the network through digital signatures, multiple copies of messages, etc. Or, it may be provided by the clock distribution and estimation algorithms.

Faults are modeled by removing components of the synchronization graph. The following fault types are defined:

- Node Faults: These correspond to the removal of a system vertex from the synchronization graph. Examples of this type of fault are dead nodes, nodes isolated by communication failures, and nodes with faulty clocks.

- Edge Faults: These correspond to the removal of an edge from the synchronization graph. Communication failures often fall into this type. Since the effects of these faults are less than the effects of the removal of either endpoint, they will not be considered further.

- Group Faults: These correspond to the removal of a group vertex from the synchronization graph. Faults which prevent the distribution of clock information fall under this type. If the distribution method is fault-tolerant, these faults are generally the consequence of multiple node faults. A number of node faults within a small part of the synchronization graph may mean that the maximum skew between members of a nearby synchronization group may no longer be guaranteed.

A group fault which is caused by the presence of node faults is called an *induced* fault. There is nothing wrong with the synchronization group itself, its non-faulty members may continue to estimate each other's clock values, but the guarantee of a maximum skew of $\hat{\delta}$ between members no longer holds.

4.4.2 Determining Fault-Tolerance

The system is considered to be synchronized as long as the synchronization graph is connected. The synchronization graph can become disconnected solely because of faults, or through a combination of faults and induced faults.

Connectedness of the Synchronization Graph

Since each fault corresponds to the removal of a component of the synchronization graph, multiple faults may disconnect the graph. The number of faults the multicomputer can withstand is therefore limited by the minimum number of faults which can disconnect the synchronization graph.

The minimum cut, or *connectedness*, of a graph is a well-known problem from graph theory. It can be solved through use of a max-flow algorithm in conjunction with the max-flow min-cut theorem. A straightforward, linear-time transformation can change the graph to an instance of max-flow min-cut. Let E and V be the number of edges and vertices in the transformed graph, then the max-flow problem can be solved in $O(EV \log(V^2/E))$ [13].

Collapse of Synchronization Groups

A synchronization group is said to have *collapsed* if it can be shown that the maximum skew after synchronization between two of its non-faulty members, as calculated in Section 4.3, is greater than $\hat{\tau}$. The collapse of a synchronization group is an induced group fault.

The problem with induced faults is they may cascade. An induced group fault can induce further group faults, which can induce more group faults, until all groups have collapsed. Exact computation of the minimum number of faults required to cause such a cascade is a difficult problem, but it is possible to compute the minimum number of node faults needed to induce a group fault, and the minimum number of group faults needed to induce more group faults. A combination of these two numbers gives a good estimate for a lower bound on the number of faults needed to produce a cascade.

To determine the number of faults needed to collapse a synchronization set, one must list all the different synchronization areas, then check each to find the minimum number of faults to cause a synchronization group to collapse. These are the same areas which are checked when computing maximum skew. If there is more than one synchronization area to consider, the search should start with the ones whose synchronization groups have

maximum skews already close to $\hat{\tau}$, or have small values of either e_i or e_j.

Finding the minimum fault set to collapse a synchronization group requires searching all fault sets of the synchronization area. While the number of fault sets within a synchronization area is much less than the number in the entire multicomputer, it may still be rather large. The approach taken here is a simple tree search. Each node of the tree represents a fault set. The root of the tree corresponds to the empty fault set, and the children of a node correspond to the fault sets created by adding a single fault to the parent fault set. Thus, nodes a depth of one in the tree correspond to fault sets of size one, nodes a depth of two correspond to fault sets of size two, and so on. Each node also has a corresponding maximum skew, which is found by inserting that node's fault set and computing the maximum skew. The search tries to find the node closest to the root which has a maximum skew greater than $\hat{\delta}$. The implementation here uses a depth-first search, though other types of search could be used. A breadth-first search in particular would parallelize well, making it a good choice for implementation on a multiprocessor. Any of the following discussion applies equally well to a breadth-first approach.

Once a fault set has been found which collapses a synchronization group, only the nodes above it in the tree have to be searched. The number of nodes in the tree increases exponentially with depth. It is therefore important to find small fault sets quickly, as it will greatly reduce the search time. When searching the children of a node, one should start with those whose fault sets are thought most likely to be subsets of the minimum fault set. Two strategies were employed to do this. The first strategy was to rate each fault set according to the types of faults it contained. Fault sets which contained group faults, and/or node faults involving intersection vertices or 1-vertices, were searched first. The second strategy was to compute the maximum skew for each fault set, and search those with the highest skews first. The first strategy was quickly abandoned, it would take days to find sets which the second would find in minutes. In fact, the second strategy usually found the smallest fault set almost immediately. The first strategy seemed to fail because while the favored fault types caused great increases in maximum skew at first, they tended to mask one another's effects (especially in the case of group faults), and faults added later would have little or no effect on maximum skew.

Even though the search found the minimum fault set quickly, a large number of nodes may still have to be searched. In one of the examples below, the synchronization area contains in excess of 40 system vertices, while the minimum set is 13 node faults. To verify

that this set is the minimum, one has to search all nodes a depth of 12 in the tree, a total of over 5 billion nodes. To allow the search to finish within a reasonable amount of time two strategies were employed which caused the search to skip those nodes thought to be unlikely to yield a minimum fault set. First, if several children of the current node have identical skews, search only one of them, the rest are assumed to be "equivalent". Second, do not search any children which do not have a greater maximum skew than their parent. These strategies greatly reduced the search time, from 22 days to less than 24 hours, in one case.

Any strategies to reduce the number of nodes searched may cause the search to overlook the minimum fault set. No such cases were encountered. In fact, the first set found was almost always the minimum set. However, it is wise to consider the smallest set found to be only an estimate of the actual fault-tolerance. As shown in the examples, it is possible to intentionally underestimate the fault-tolerance by changing ε and $\hat{\tau}$. This can be used to reassure oneself that the system has the desired fault-tolerance. It should also be possible to intentionally underestimate fault-tolerance by ignoring 2-SPs. Though this strategy was not tried, it should speed up the search considerably, and could be used for systems where the minimum fault set is too large for the search to find within a reasonable time.

4.4.3 Examples

Faults have the effect of increasing the value on the right-hand side of Equation (4.7). A $\hat{\tau}$ which worked when there were no faults may be too small when faults are present. The value of $\hat{\tau}$ must be chosen large enough so that there is considerable difference between the two sides of the inequality in Equation (4.7) when no faults are present, so that Equation (4.7) is still satisfied when faults are present. Each example below requires only a single synchronization area be considered, because the synchronization graphs are such that there exists only one unique synchronization area. This is partly because of the homogeneous nature of the systems under consideration, and partly because such graphs are easier to deal with.

The first example is the 16-node hypercube of Figure 4.1. For a best-case estimate of fault-tolerance it is assumed there is no estimation uncertainty and continuous synchronization, i.e., $\varepsilon = 0$ and $\hat{\tau} = \hat{\delta}$. In order to collapse a synchronization group a minimum of 5 node faults or 3 group faults is needed. Fault-tolerance decreases as ε increases and increases as $\hat{\tau}$ increases. If ε is increased to $.1\hat{\delta}$ and $\hat{\tau}$ is decreased to $.9\hat{\delta}$, either 3 node faults

or 1 group fault will collapse a synchronization group. Thus, 3 node faults may induce a cascade of group faults which will engulf the system.

For a second example consider a 16×16 square mesh, wrapped on the edges to provide a homogeneous system. The method suggested in Section 4.3.5 is used to define 32 synchronization groups of 16 members each, one group for each row of the mesh, and one group for each column. If $\varepsilon = 0$ and $\hat{\tau} = \hat{\delta}$, either 14 node faults or 9 group faults are needed to collapse a synchronization group. Increasing ε and decreasing $\hat{\tau}$ to the more realistic values of $\varepsilon = .1\hat{\delta}$ and $\hat{\tau} = .9\hat{\delta}$, has a considerable effect on fault-tolerance. In this case, either 6 node faults or 1 group fault are needed to collapse a synchronization group. So at least 6 node faults are needed to induce a cascade of group faults.

The fault-tolerance of the previous example can be improved if more groups are used. Consider a 256-node hypercube with 64 synchronization groups of 16 members each. The hypercube has an 8-bit address, $abcdefgh$. Each synchronization group is a 4-bit addressable subcube, where a subcube is defined by fixing four bits of the address and allowing the other four bits to vary. If x indicates a "don't care" position in the address the 64 subcubes are defined as follows: 16 of the form $abcdxxxx$, 16 of the form $xxxxefgh$, 16 of the form $abxxxxfg$, and 16 of the form $xxcdefxx$. The extra groups greatly improve fault-tolerance. When $\varepsilon = 0$ and $\hat{\tau} = \hat{\delta}$, the search algorithm did not terminate in over 4 weeks. The smallest set found was 27 node faults. The search terminates if ε is increased and $\hat{\tau}$ is decreased. When $\varepsilon = .1\hat{\delta}$ and $\hat{\tau} = .9\hat{\delta}$, either 13 node faults or 2 group faults are needed to collapse a synchronization group. A further reduction of $\hat{\tau}$ still allows a number of faults to be tolerated. If $\hat{\tau} = .8\hat{\delta}$ then 8 node faults are required to collapse a synchronization group.

The final example considers two 1024-node multicomputers, a 10-cube, and a $32\times$ 32 wrapped square mesh. For the 10-cube define 64 synchronization groups, 32 5-cubes of the form $abcdexxxxx$, and 32 5-cubes of the form $xxxxxfghij$. For the square mesh use the method in Section 4.3.5 and define each row and each column to be a synchronization group. Each of these two systems has 64 groups of 32 members each. In fact, the synchronization graphs for the two systems are isomorphic, so their fault-tolerances are identical. These systems are too large to solve easily when $\varepsilon = 0$ and $\hat{\tau} = \hat{\delta}$. If ε is increased to $.1\hat{\delta}$ and $\hat{\tau}$ is decreased to $.9\hat{\delta}$ each system requires 10 node faults or 1 group fault before a synchronization group can collapse. A further reduction of $\hat{\tau}$ to $.8\hat{\delta}$ still requires 5 node faults in order to collapse a synchronization group. While a set of 10 faults may not seem like much in a 1024-node system, note that each node gathers information on only 62 other

clocks, and that the 10 faults must be confined to a relatively small portion of the system. Also, fault-tolerance can be improved by increasing the number of groups, as shown in the previous examples.

The above examples also show the extent of the reduction in the number of estimates. The reduction in estimates can be determined by comparing the number of nodes in a node's synchronization set to the number of nodes in the system. In the 16-node hypercube example each node has a synchronization set of 6 nodes, so each node need only communicate with 6 other nodes in order to synchronize. This is compared with the 15 nodes of standard algorithms, a two-fold reduction. For the 16×16 square mesh sees a reduction of 255/30 or more than 8-fold, doubling the number of synchronization groups to improve fault-tolerance reduces the savings to only 4-fold. In the last example, each node need only communicate with 62 out of 1023 other nodes in order to synchronize, over a 16-fold decrease in the number of synchronization messages.

The examples also show how well fault-tolerance is maintained, and how it can be adjusted. The examples also demonstrate a method for dealing with the difficulty in determining absolute fault-tolerance. By increasing ε decreasing $\hat{\tau}$, or reducing the number of synchronization groups, the fault-tolerance of the multicomputer is reduced, and so is the time required to find a minimum fault set. By adjusting these parameters, not only will one find a lower bound for fault-tolerance, but by analyzing how each parameter affects fault-tolerance one may be able to estimate fault-tolerances which the search would take too long to find.

4.5 Summary

As system sizes get larger, it becomes very expensive for each node to make an estimate of every other node in the system. This makes peer synchronization algorithms difficult to use.

This chapter described a peer adjustment algorithm which requires nodes make estimates for only a subset of the nodes in a system, yet still maintains synchronization. Groups are defined and each node makes estimates of those nodes which belong to the same groups. The principle features of the algorithm are the following:

- The total number of estimates made by each node is greatly reduced.
- Within groups synchronization is tighter than in the system overall. Assigning coop-

erating tasks to nodes in a common group provides them with the synchronization they need. Tasks that do not require tight synchronization can be assigned to nodes in different groups.

- Fault tolerance is also provided. Limiting the number of nodes for which each node makes estimates also reduces the likelihood that any particular node will be affected by a fault.

CHAPTER 5

EFFICIENT PROBABILISTIC ESTIMATION

Probabilistic estimation algorithms are different from other estimation algorithms in that the uncertainty of their estimates is not limited by the maximum variability in message delay. They can, in theory, reduce estimate uncertainty to any desired level. The price paid for this feature is a tremendous increase in the number of synchronization messages per estimate. This increase in network traffic makes probabilistic estimation impractical in large systems.

The first published probabilistic estimation algorithm is found in [11]. This paper describes a simple master/slave synchronization algorithm employing a probabilistic estimation algorithm. Distribution of clock information is done by private communication between master and slave. Since the adjustment algorithm is a master/slave algorithm, efficiency in the distribution of clock information is not a major concern, since only a few nodes are distributing clock information.

A second probabilistic estimation algorithm appeared in [2]. This paper also describes a simple master/slave synchronization algorithm employing a probabilistic estimation algorithm. Distribution of clock information can be done via private communication from master to slave, or via broadcasts from the master. Again, the master/slave adjustment algorithm limits the amount of clock information that must be distributed, so efficiency in distribution is not an issue.

The probabilistic estimation algorithms in [11] and [2] are the basis of all other known probabilistic estimation algorithms. NTP [31, 32], while not billed as a probabilistic algorithm, effectively uses the algorithm in [11] within a hierarchical master/slave system.

In [36] an attempt is made to use the estimation algorithm of [2] with a peer adjustment algorithm. A highly-efficient broadcast mechanism is described in order to reduce the cost of distributing clock information. The algorithm proceeds as a series of rounds. In each

round each node gathers the timestamps it received in the previous round, discards any duplicates, puts them all into a message, timestamps it, and sends a copy to each of a predetermined set of nodes. While the number of messages and bytes transmitted is greatly reduced, in even a moderately large system the synchronization algorithm literally swamps the system while it is in operation. Each round a node receives a number of messages, each full of timestamps, processes all the timestamps it receives, discards many (if not most) of them, builds a new synchronization message, and sends it to a number of nodes. The heavy load imposed by this algorithm can impair normal system operations.

This chapter describes an efficient network algorithm for distributing clock information, and two probabilistic estimation algorithms designed to work with it. Clock information is distributed globally, and the estimation algorithms generate estimates for all nodes, allowing peer synchronization of the system. However, unlike [36], synchronization messages are not sent all at once, nor is large amounts of information discarded, so system operations are not impaired while the synchronization algorithm is running. Extensive analysis is done to verify the effectiveness of the algorithm, and simulation results are presented to back up the analysis. Fault-tolerance is considered also, to show that it has not been sacrificed for the sake of efficiency.

The chapter starts with the introduction of the distribution algorithm. Then, the two probabilistic estimation algorithms are described, analyzed, and simulated. Several possibilities for improving the clock distribution algorithm are then considered. Finally, the fault-tolerance of the distribution algorithm is discussed.

5.1 Synchronization Message Paths

In order to estimate α_{jli}, N_i's probabilistic estimation algorithm must generate a sequence of high-uncertainty guesses of α_{jli}, and combine them into a single low-uncertainty estimate. Each guess of α_{jli} must be made using clock information gathered independently from the clock information used to make any other guess of α_{jli}, or the independence assumptions upon which probabilistic estimation algorithms rely are lost.

Distributing clock information is therefore a sequence of independent *inquiries*, each inquiry generating at most a single guess for any α_{jli}. Each inquiry involves sending one or more synchronization messages along pre-specified paths. The paths taken determine how many nodes can participate in a single inquiry, and how efficient the inquiries are.

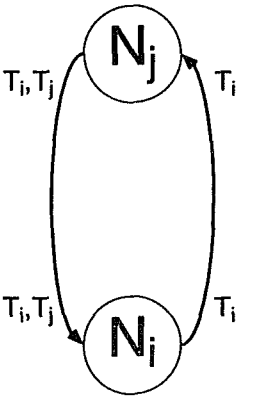

Figure 5.1: Single-node inquiry

5.1.1 Single-Node Inquiry

The probabilistic estimation algorithms in [3, 11] use the *single node* inquiry shown in Figure 5.1, which allows a single node to make a guess of only one other node[1]. N_i wishes to guess its skew with respect to N_j, so it sends a synchronization message to N_j and affixes its timestamp, T_i. Some time later the message arrives at N_j, with T_i still attached. N_j prepares a new synchronization message containing T_i, adds its own timestamp, T_j, and sends the message to N_i. Later, the message arrives at N_i, with both T_i and T_j attached, and the arrival time T_i^a is noted. A guess of N_j's skew with respect to N_i can be made with a simple computation involving T_i, T_j, and T_i^a.

Each guess therefore requires two synchronization messages be sent, one by N_i and one by N_j. For a single node to guess its skew with respect to every other node in an n-node system, $2n-2$ synchronization messages containing $3n-3$ timestamps must be sent. For every node to make a guess of every other node, $2n^2-2n$ synchronization messages containing $3n^2-3n$ timestamps must be sent. This is for only a single guess, if, as is usually the case with probabilistic estimation, each node must make multiple guesses, more inquiries will have to be done. The number of guesses needed varies with the node making the guess and the node whose skew is being guessed, but tends to increase with the distance between the two. However, if g is the average number of guesses a node must make for every other node, then a total of $2gn^2-2gn$ messages, containing $3gn^2-3gn$ timestamps, must be sent. When one considers that in order for probabilistic estimation to work properly, all these messages must be sent within a fairly short interval, one begins to understand why master/slave clock organization is popular amongst those using probabilistic estimation.

[1] [3] actually proposes a somewhat simpler inquiry that requires only a single synchronization message, but the inquiry shown in Figure 5.1 may be used when network delays are not known or unpredictable

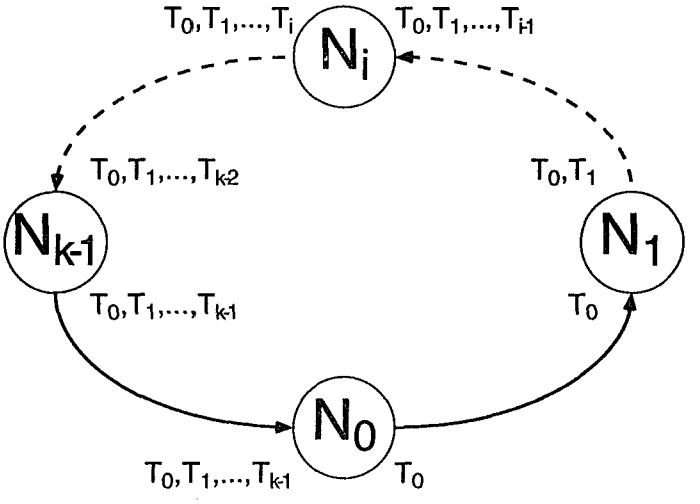

Figure 5.2: k-node inquiry

5.1.2 Double-Node Inquiry

An obvious way to improve the single-node inquiry described above is to make it a double-node inquiry by having N_i send another synchronization message to N_j containing T_j and adding its own timestamp T_i'. N_j notes the arrival time, T_j^a, and a simple calculation involving T_j, T_i', and T_j^a, produces a guess of N_i's skew with respect to N_j. The result is two guesses, from three synchronization messages, containing a total of five timestamps. A total of $3(n^2-n)/2$ synchronization messages, containing $5(n^2-n)/2$ timestamps, must be sent in order for each node to be able to make a guess of every other node in the system. The number of synchronization messages has been cut by one-quarter, and the number of timestamps has been cut by one-sixth.

5.1.3 k-Node Inquiry

Since some savings are realized by going from a single-node inquiry to a double-node inquiry, it is reasonable to ask if further savings might be realized by adding more nodes and generalizing the inquiry to k nodes.

The generalization of inquiries to k nodes is show in Figure 5.2. To simplify notation it is assumed N_0 sends the first synchronization message, and N_i, $1 \leq i \leq k-1$, sends its synchronization message to N_{i+1}. Node numbers can be re-mapped if this is not the case. The k-node inquiry begins when N_0 sends a synchronization message, containing timestamp T_0, to N_1. After it arrives, N_1 sends a synchronization message containing both T_0 and its own timestamp, T_1, to N_2. The process continues in this manner, with N_i receiving

a synchronization message containing timestamps $T_0,\ldots,T_{i-1}$ from N_{i-1}, and sending a synchronization message containing timestamps $T_0,\ldots,T_i$ to N_{i+1}. Finally, N_{k-1} sends its synchronization message to N_0. As before, N_0 notes the arrival time of the message, T_0^a. Then, simple calculations involving $T_0,\ldots,T_{k-1}$, and T_0^a produce guesses of $\alpha_{1|0}$ through $\alpha_{k-1|0}$.

At this point, only N_0 has made any guesses. However, one can continue much as with double-node inquiry. N_0 prepares a synchronization message containing $T_1,\ldots,T_{k-1}$, adds its own timestamp, T_0', and sends it to N_1. When it arrives, N_1 notes the arrival time, T_1^a, and then N_1 can make guesses of $\alpha_{2|1}$, ..., $\alpha_{k-1|1}$ and $\alpha_{0|1}$. N_1 then prepares a synchronization message containing $T_2,\ldots,T_{k-1}$ and T_0', adds its timestamp, T_1', and sends it to N_2. In general, N_i receives a synchronization message of the following form:

$$T_i,\ldots,T_{k-1},T_0',\ldots,T_{i-1}'.$$

It guesses the skew of the other $k-1$ nodes with respect to itself, then prepares a synchronization message with the following form:

$$T_{i+1},\ldots,T_{k-1},T_0',\ldots,T_{i-1}'.$$

It then adds its timestamp, T_i', and sends it to N_{i+1}.

Each node will send two synchronization messages, except N_{k-1}, which need not send a second synchronization message to N_0. Therefore, the total number of synchronization messages sent in a single inquiry is:

$$messages = 2k-1 \tag{5.1}$$

The first $k-1$ messages will contain a total of $k(k-1)/2$ timestamps, the final k messages will contain a total of k^2 timestamps. The total number of timestamps sent in a single inquiry is therefore:

$$timestamps = \left(3k^2-k\right)/2 \tag{5.2}$$

If $k=n$, then every node can make a guess of every other node by sending only $2n-1$ synchronization messages containing $(3n^2-n)/2$ timestamps, an n-fold reduction in the number of synchronization messages, and a nearly fifty percent reduction in the number of timestamps, compared to single-node inquiry. However, it is only fair to mention that an inquiry involving such a large number of nodes is likely to generate rather poor estimates. Analysis later in this chapter will show how much the estimates suffer.

5.1.4 Forward and Backward Inquiries

Since T_1 is created well before T_{k-1}, it will be much "older" when N_0 uses it to make its guesses. As a result, N_0's guess of N_1 will have a greater uncertainty than its guess of N_{k-1}. The obvious solution is to reverse the order and direction of synchronization message sends, N_0 will send first to N_{k-1}, which sends to N_{k-2} and so on, until N_1 sends to N_0. This way T_1 is "fresher" than T_{k-1} when it arrives at N_0, and N_0's guess of N_1 will have a lower uncertainty than its guess of N_{k-1}.

Inquiries done as described in Section 5.1.3, where N_i sends synchronization messages to $N_{i+1 \bmod k}$, are called *forward* inquiries. Inquiries done in the opposite direction, where N_i sends synchronization messages to $N_{i-1 \bmod k}$ are called *backward* inquiries. Use of both forward and backward inquiries is not necessary, but is a simple way to improve estimation algorithm performance, without any corresponding cost increase in distributing clock information (especially in systems with bi-directional communications.)

5.1.5 Using Arrival Timestamps

In cases where the delay at individual nodes may be long or highly variable, it is often useful to modify the k-node inquiry by having each node include its arrival timestamp in the synchronization message. That is, when N_i prepares the synchronization message, it will include not only timestamps from the synchronization message it just received, but T_i^a as well. The first synchronization message N_i receives will thus have the form:

$$T_0, T_1^a, T_1, \ldots, T_{i-1}^a, T_{i-1}.$$

And, the second synchronization message received by N_i will have the form:

$$T_i, T_{i+1}^a, T_{i+1}, \ldots, T_0^{a\prime}, T_0', \ldots, T_{i-1}^{a\prime}, T_{i-1}'.$$

The addition of the arrival timestamps allows nodes to estimate the delays at intermediate nodes more accurately, at the expense of added complexity and larger synchronization messages.

5.2 Interval-Oriented Estimation

When N_i receives the second synchronization message, it knows that all timestamps in the message were generated in between T_i, when it sent its first synchronization message, and

T_i^a, the arrival of the second synchronization message. Furthermore, if the minimum time for N_i's timestamp to reach N_j is $\underset{i\to j}{d}$, then N_i can be certain that N_j's timestamp was not created before time $T_i + \underset{i\to j}{d}$ on C_i^S. A similar argument produces an upper bound of $T_i^a - \underset{j\to i}{d}$ on the time on C_i^S when N_j created its timestamp. The bounds can be further refined by taking advantage of the arrival timestamps (if present), and by taking into account possible clock drift during the inquiry.

Each guess is therefore an *interval* containing the time on C_i^S when the node created its timestamp. The uncertainty of a guess is half its width, and guesses can be combined by intersecting them. The interval resulting from the intersection of a number of guesses will be smaller than any of the individual guesses, and its expected size decreases as the number of guesses increases. To make the estimate, one need only intersect enough guesses to make the width of the resulting interval small enough to have the desired uncertainty.

This estimation algorithm is much like the one described by Cristian [11]. However, the algorithm in [11] is tailored to single-node inquiry. No combination of guesses is done. Guesses are simply discarded if their uncertainty is too high, and the algorithm continues to make guesses until it finds one whose uncertainty is low enough. The algorithm in [30] does intersection of intervals, but it is by no means a probabilistic algorithm.

5.2.1 Calculation of Intervals

To simplify notation, and without loss of generality, consider only the calculation of intervals from the point of view of N_0. All other nodes will do the same calculation, with the node numbers re-mapped appropriately. If arrival timestamps are not included in the synchronization messages, then $T_i^a \equiv T_i$ for $0 < i < k$.

The following definitions are used in the computations below:

N_k: The node to which N_{k-1} sends synchronization messages during forward inquiries, and from which it receives synchronization messages during backward inquiries. Technically, this is N_0, but notation is simplified (in particular, modulus operations are eliminated) if this convention is used.

T_0, T_k: For forward inquiries, T_0 is the timestamp created by N_0 and T_k is the arrival timestamp created by N_k. For backward inquiries, T_k is the timestamp created by N_k and T_0 is the arrival timestamp created by N_0. Note that for forward inquiries, $T_k^a = T_k$ while T_0^a is undefined, and for backward inquiries $T_0^a = T_0$ while T_k^a is

undefined.

d_A: The *actual* time between the creation of T_0 and T_k, measured in the external reference frame. This is the time measured by an individual with a "perfect" clock who is observing the system.

$\underset{i\to i+1}{d}$: The *minimum* time between the creation of T_i and T_{i+1}^a, $0 \le i < k$, during a forward inquiry. Again, this time is measured in the external reference frame. Define $\underset{i\to j}{d}$, $0 \le i < j \le k$ as a sum of the individual minimums:

$$\underset{i\to j}{d} \equiv \sum_{h=i}^{j-1} \underset{h\to h+1}{d}$$

These minimums are *absolute* minimums, and do not depend on the actual delays. It is physically impossible for the time between the creation of T_i and T_{i+1}^a to be less than $\underset{i\to i+1}{d}$. Analogous definitions of $\underset{i\to i-1}{d}$ and $\underset{i\to j}{d}$ exist for backward inquiries.

$\underset{i\to i+1}{x}$: The *excess* time between the creation of T_i and T_{i+1}^a, $0 \le i < k$, in a forward inquiry. Therefore, $\underset{i\to i+1}{d} + \underset{i\to i+1}{x}$ is the *actual* time between the creation of T_i and T_{i+1}^a. As for $\underset{i\to j}{d}$, $\underset{i\to j}{x}$, $0 \le i < j \le k$, is the sum of the individual excesses:

$$\underset{i\to j}{x} \equiv \sum_{h=i}^{j-1} \underset{h\to h+1}{x}$$

Again, analogous definition of $\underset{i\to i-1}{x}$ and $\underset{i\to j}{x}$ exist for backward inquiries.

W_i: The *measured* time between the creation of T_i^a and T_i, i.e., $W_i = T_i - T_i^a$. This is called the *node wait* at N_i. The sum of a series of node waits in a forward inquiry, $\underset{i\to j}{W}$, is defined as

$$\underset{i\to j}{W} \equiv \sum_{h=i+1}^{j} W_h$$

for $0 \le i < j \le k$. The definition for backward inquiries is analogous. Note that $\underset{i\to j}{W}$ includes W_j but not W_i, in this sense it is the total wait between the creation of N_i's and N_j's timestamps. If arrival timestamps are not included in the synchronization messages, all node waits are zero.

w_i: The *actual* time between the creation of T_i^a and T_i, as measured in the external reference frame. This is called the *actual node wait* at N_i. The sum of a series of actual node waits in a forward inquiry, $i \xrightarrow{w} j$, is defined as

$$i \overset{w}{\rightarrow} j \equiv \sum_{h=i+1}^{j} w_h.$$

for $0 \leq i < j \leq k$. Again, the definition for backward inquiries is analogous. As in the case of node waits, actual node waits are defined to be zero if arrival timestamps are not included in the synchronization messages.

The values of ${}_{i\rightarrow i+1}d$ and ${}_{i\rightarrow i-1}d$ are determined during system design, and thus are available to the synchronization algorithm. All other actual delays are not available, but can be underestimated or overestimated (as appropriate) using the clock values available.

The following derivation is for forward inquiries. The derivation for backward inquiries is similar. To start, consider the relationship between actual delays:

$$d_A = {}_{0\rightarrow k}d + {}_{0\rightarrow k}x + 0 \overset{w}{\rightarrow} k$$

Because each ${}_{i\rightarrow i+1}d$ is an absolute minimum, it follows that ${}_{i\rightarrow i+1}x \geq 0$. So for $0 \leq i < k$,

$$d_A - {}_{0\rightarrow k}d - 0 \overset{w}{\rightarrow} k \geq {}_{i\rightarrow k}x \geq 0 \tag{5.3}$$

Let t_k indicate the time in the external reference frame when T_k was created, i.e., $C_0^S(t_k) = T_k$. $C_i^S(t_k)$ is then time on N_i's synchronization clock when N_0's synchronization clock had value T_k. Then for any $0 < i < k$,

$$C_i^S(t_k) \in \left[T_i + \left({}_{i\rightarrow k}d + {}_{i\rightarrow k}x + i \overset{w}{\rightarrow} k\right)(1-\varrho),\ T_i + \left({}_{i\rightarrow k}d + {}_{i\rightarrow k}x + i \overset{w}{\rightarrow} k\right)(1+\varrho)\right] \tag{5.4}$$

While this interval is quite small, and would therefore produce a guess with a very low uncertainty, its bounds cannot be calculated by N_0, since N_0 does not know the value of ${}_{i\rightarrow k}x$ for any i. The ${}_{i\rightarrow k}x$ values must be underestimated for the lower bound of the interval in Equation (5.4), and overestimated for the upper bound of the interval in Equation (5.4). Equation (5.3) provides the necessary underestimates and overestimates, yielding the following:

$$C_i^S(t_k) \in \left[T_i + \left({}_{i\rightarrow k}d + i \overset{w}{\rightarrow} k\right)(1-\varrho),\ T_i + \left(d_A - {}_{0\rightarrow i}d - 0 \overset{w}{\rightarrow} i\right)(1+\varrho)\right]$$

This leaves d_A and the actual node waits as the unknown quantities. A bound can be placed on d_A using ϱ, namely $d_A(1-\varrho) \leq T_k - T_0$, or $d_A \leq (T_k - T_0)/(1-\varrho)$. Similarly, w_j can also be bounded, $w_j(1+\varrho) \geq W_j$, or $w_j \geq W_j/(1+\varrho)$. Substituting gives

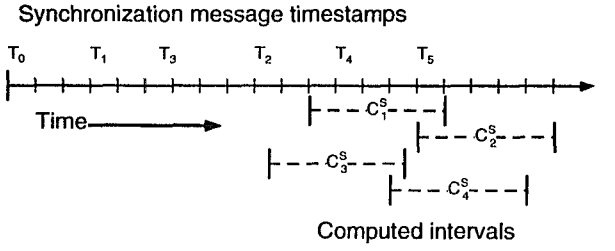

Figure 5.3: Intervals calculated from timestamps

$$C_i^S(t_k) \in \left[T_i + \left(d_{i\to k} + \frac{W_{i\to k}}{1+\varrho}\right)(1-\varrho),\ T_i + \left(\frac{T_k - T_0}{1-\varrho} - d_{0\to i}\right)(1+\varrho) - W_{0\to i}\right] \tag{5.5}$$

Somewhat more useful than the interval of Equation (5.5) is the corresponding *skew interval.* To get an interval containing the skew between C_i^S and C_0^S, subtract T_k from each endpoint:

$$\alpha_{i0} \in \left[T_i - T_k + \left(d_{i\to k} + \frac{W_{i\to k}}{1+\varrho}\right)(1-\varrho),\ T_i - T_k + \left(\frac{T_k - T_0}{1-\varrho} - d_{0\to i}\right)(1+\varrho) - W_{0\to i}\right] \tag{5.6}$$

The derivation for backward inquiries proceeds exactly as above. Or, one can simply re-map the node numbers in Equation (5.6). Either way, the resulting interval for backward inquiries is the following:

$$\alpha_{i0} \in \left[T_i - T_0 + \left(d_{i\to 0} + \frac{W_{i\to 0}}{1+\varrho}\right)(1-\varrho),\ T_i - T_0 + \left(\frac{T_0 - T_k}{1-\varrho} - d_{k\to i}\right)(1+\varrho) - W_{k\to i}\right] \tag{5.7}$$

Figure 5.3 shows what might happen when $k = 5$. Time increases to the right, and T_0 through T_5 at the top of the figure represent the relative values of the timestamps. T_0 was created by N_0 when it sent the first synchronization message, and T_5 is T_0^a for the synchronization message sent by N_4. Arrival timestamps were not included in the synchronization messages. The intervals for $C_1^S(t_5)$ through $C_4^S(t_5)$ are computed by N_0 using Equation (5.5), and are shown beneath the timestamps on the right side of the figure. The intervals determine the range of possible values for their respective clocks at the time N_0 created T_5. For instance, since the intervals for C_2^S and C_3^S do not overlap, C_2^S and C_3^S cannot have the same value, and the minimum skew between them is the difference between the lower bound of C_2^S's interval and the upper bound of C_3^S's interval. Also, since the interval for C_3^S does not contain T_5, the minimum skew between C_0^S and C_3^S is the difference between T_5 and the upper bound of C_3^S's interval.

5.2.2 Convergence

All probabilistic algorithms assume that the delays are independent random variables. That is, the delay distributions for $\underset{i \to i+1}{x}$ and w_i depend neither on each other nor on the values of either $\underset{j \to j+1}{x}$ or w_j, $i \neq j$. This assumption is made partly because analysis assuming dependent delays is almost impossible, and partly because testing and experience have shown it to yield correct results in most cases. Independence implies that each time the inquiry is done the delays will be different, and the resulting set of skew intervals will be different. Each skew interval is a guess, whose uncertainty is half the width of the interval. After q inquiries have been done, there will be q guesses for each α_{i0}. Since α_{i0} must be in all q guesses, it will be in their intersection, and their intersection will be yet another interval. As q increases, the width of the intersection interval tends to decrease, with the rate of decrease slowing as the interval becomes smaller. This process is known as *convergence.*

Convergence can be sped up by sending messages in both directions. Some understanding of how can be gained from Figure 5.3. This figure was drawn as if C_0^S, C_1^S, and C_4^S were tightly synchronized, while C_2^S was running ahead, and C_3^S was running behind. Note that even though at T_5 on C_0^S both C_1^S and C_4^S should also read approximately T_5, T_5 is not centered in either interval. T_5 appears in the upper half of the interval for C_1^S, and in the lower half of the interval for C_4^S. An examination of the derivation of Equation (5.5) shows that this is representative of a general trend. Much of the width of the interval is due to the substitution for $\underset{i \to k}{x}$ using Equation (5.3). The lower bound of the interval substitutes 0 for $\underset{i \to k}{x}$, for small i this will usually be a considerable underestimate, for i close to k this is a close approximation. Thus, the lower bound of the skew interval will be considerably less than the actual skew when i is small, and fairly close to the actual skew when i is close to k. The upper bound has the reverse situation. The upper bound substitutes $d_A - \underset{0 \to k}{d} - w_1 k - 1$ for $\underset{i \to k}{x}$, which will be a close approximation when i is small, but far too large when i is close to k. The upper bound of the skew interval will be close to the actual value when i is small and considerably greater than the actual skew value when i is close to k.

The situation is reversed for backward inquiries. The lower bound of intervals is close to the actual skew value when i is small, and considerably less than the actual skew value when i is near k. The upper bound of intervals is considerably greater than the actual skew value when i is small, and close to the actual skew value when i is near k. Obviously, intersecting an interval generated by a forward inquiry with one generated by a backward inquiry will result in an interval much smaller than either type of inquiry is likely to generate on its

own (at least for intervals where i is either small or close to k.) Convergence can therefore be greatly sped up by alternating forward and backward inquiries.

5.2.3 Analysis

The estimation algorithm can easily tell if any more inquiries need to be done — it simply computes the average uncertainty of the current estimates and stops if it is less than ε. However, since the total number of inquiries sent determines the total number of messages and timestamps sent, one should know beforehand the maximum number of inquiries that are going to be needed. It would be nice to have a function, $\Phi(q)$, whose value was the probability that only q inquiries will have to be sent. Unfortunately, such a function is nearly impossible to compute. While the delays themselves are independent variables, the intervals are functions of the delays, and are not independent of one another. Finding $\Phi(q)$, therefore, necessitates the convolution of a number of dependent random variables.

However, all is not lost. It is possible to characterize the width of a *single* intersection interval. The width of the intersection interval for N_i after j inquiries, Ω_i^j, is a function of the delays. If the distributions of the delays are known, the distribution of Ω_i after any number of inquiries can be determined. From the distribution one can determine the probability that the width of the interval is below a specified value (such as 2ε).

It is assumed that the estimation process consists of $2q$ inquiries, numbered 0 through $2q-1$. Inquiries alternate between forward and backward, even-numbered inquiries are forward and odd-numbered inquiries are backward. Consecutive inquiries are assumed to start λ clock ticks apart. For the most part, this analysis will concentrate on forward inquiries, mentioning backward inquiries only for important results, where the corresponding equation for backward inquiries will be presented also.

The focus of this analysis are random variables $\mathcal{X}_{0\to i}^h$ and $\mathcal{X}_{i\to k}^h$, the values of $\underset{0\to i}{x}$ and $\underset{i\to k}{x}$ for inquiry h. The distribution and density functions of these variables are assumed to be known, and do not vary with h. Furthermore, $\mathcal{X}_{i\to j}^h$ and $\mathcal{X}_{i'\to j'}^h$ are assumed to be independent if either $0 \le i < j \le i' < j' \le k$ or $0 \le j < i \le j' < i' \le k$ holds, and $\mathcal{X}_{i\to j}^h$ and $\mathcal{X}_{i\to j}^{h'}$ are independent if $h \ne h'$. Other random variables of minor importance are $\mathcal{W}_{i\to j}^h$, and the value of $\underset{i\to j}{w}$ for inquiry h. The distribution of the node waits need not be known, but a good bound on the maximum total node wait should be available.

Let T_i^h be N_i's timestamp from inquiry h. The following inequalities will hold:

$$T_k^h \geq T_0^h + \left(\underset{0\to k}{d} + \underset{0\to k}{\mathcal{X}^h} + \underset{0\to k}{\mathcal{W}^h}\right)(1-\varrho) \tag{5.8}$$

$$T_k^h \leq T_0^h + \left(\underset{0\to k}{d} + \underset{0\to k}{\mathcal{X}^h} + \underset{0\to k}{\mathcal{W}^h}\right)(1+\varrho) \tag{5.9}$$

$$T_i^h \geq T_0^h + \alpha_{i0} + \left(\underset{0\to i}{d} + \underset{0\to i}{\mathcal{X}^h} + \underset{0\to i}{\mathcal{W}^h}\right)(1-\varrho) \tag{5.10}$$

$$T_i^h \leq T_0^h + \alpha_{i0} + \left(\underset{0\to i}{d} + \underset{0\to i}{\mathcal{X}^h} + \underset{0\to i}{\mathcal{W}^h}\right)(1+\varrho) \tag{5.11}$$

Let L_i^h be the value of the lower bound of the interval for N_i generated by inquiry h. The goal is to re-write the lower bound from Equation (5.6) in terms of the random variables. Unfortunately, straightforward substitution cannot be done since Equation (5.6) is written in terms of measured delays, while the random variables are actual delays. The inequalities in Equations (5.9) through (5.10) can be used to underestimate or overestimate (whichever is appropriate) some of the measured values with random variables. One would normally substitute $\underset{i\to k}{\mathcal{W}^h}(1-\varrho)$ for $\underset{i\to k}{W}$, but this substitution will not cancel out the $(1+\varrho)$ in the denominator, and will make the derivation considerably more difficult. However, since ϱ^2 is very small, $(1+\varrho)(1-2\varrho) \approx (1-\varrho)$, so $\underset{i\to k}{\mathcal{W}^h}(1-2\varrho)$ may be substituted for $\underset{i\to k}{W}/(1+\varrho)$. Performing all these substitutions, and dropping the ϱ^2 terms, results in the following:

$$\begin{aligned} L_i^h &\geq T_0^h + \alpha_{i0} + \left(\underset{0\to i}{d} + \underset{0\to i}{\mathcal{X}^h} + \underset{0\to i}{\mathcal{W}^h}\right)(1-\varrho) - T_0^h \\ &\quad - \left(\underset{0\to k}{d} + \underset{0\to k}{\mathcal{X}^h} + \underset{0\to k}{\mathcal{W}^h}\right)(1+\varrho) + \left(\underset{i\to k}{d} + \underset{i\to k}{\mathcal{W}^h}(1-2\varrho)\right)(1-\varrho) \\ &\geq \alpha_{i0} + \left(\underset{0\to k}{d} + \underset{0\to i}{\mathcal{X}^h} + \underset{0\to k}{\mathcal{W}^h}\right)(1-\varrho) - \left(\underset{0\to k}{d} + \underset{0\to k}{\mathcal{X}^h} + \underset{0\to k}{\mathcal{W}^h}\right)(1+\varrho) - 2\varrho\underset{i\to k}{\mathcal{W}^h}(1-\varrho) \\ &\geq \alpha_{i0} - \underset{i\to k}{\mathcal{X}^h}(1+\varrho) - 2\varrho\left(\underset{0\to k}{d} + \underset{0\to i}{\mathcal{X}^h} + \underset{0\to k}{\mathcal{W}^h} + \underset{i\to k}{\mathcal{W}^h}\right) \end{aligned} \tag{5.12}$$

Equation (5.12) is correct at the conclusion of inquiry h. It may no longer be correct by the time all inquiries have been completed. To fix this, a term is added to account for any possible clock drift over the course of later inquiries:

$$L_i^h \geq \alpha_{i0} - \underset{i\to k}{\mathcal{X}^h}(1+\varrho) - 2\varrho\left(\underset{0\to k}{d} + \underset{0\to i}{\mathcal{X}^h} + \underset{0\to k}{\mathcal{W}^h} + \underset{i\to k}{\mathcal{W}^h}\right) - 2\varrho\lambda(2q-h-1) \tag{5.13}$$

The bound for backward inquiries can be computed in a similar manner:

$$L_i^h \geq \alpha_{i0} - \underset{i\to 0}{\mathcal{X}^h}(1+\varrho) - 2\varrho\left(\underset{k\to 0}{d} + \underset{k\to i}{\mathcal{X}^h} + \underset{k\to 0}{\mathcal{W}^h} + \underset{i\to 0}{\mathcal{W}^h}\right) - 2\varrho\lambda(2q-h-2) \tag{5.14}$$

Since ϱ is usually small, often 10^{-5} or less, the 2ϱ terms can often be neglected for ballpark estimates. For more conservative results, one can usually replace the first 2ϱ term with a constant, β_i^L. The value of β_i^L should be an upper bound for the sum it replaces, and is usually computed as the sum of $\underset{0\to k}{d}$ and the worst-case values of the random variables.

A similar derivation can be done for the upper bound of the interval for N_i generated by inquiry h. This time the approximation $(1-\varrho)(1+2\varrho) \approx (1+\varrho)$ is used in the $(T_k^h - T_0^h)/(1-\varrho)$ term, and again the resulting ϱ^2 terms are dropped. The resulting upper bound for forward messages is:

$$U_i^h \leq \alpha_{il0} + \underset{0\to i}{\mathcal{X}^h}(1+\varrho) + 2\varrho\left(2\underset{0\to k}{d} + 2\underset{0\to k}{\mathcal{X}^h} + 2\underset{0\to k}{\mathcal{W}^h} + \underset{0\to i}{\mathcal{W}^h}\right) + 2\varrho\lambda(2q-h-1) \qquad (5.15)$$

For backward messages the upper bound is:

$$U_i^h \leq \alpha_{il0} + \underset{k\to i}{\mathcal{X}^h}(1+\varrho) + 2\varrho\left(2\underset{k\to 0}{d} + 2\underset{k\to 0}{\mathcal{X}^h} + 2\underset{k\to 0}{\mathcal{W}^h} + \underset{k\to i}{\mathcal{W}^h}\right) + 2\varrho\lambda(2q-h-2) \qquad (5.16)$$

Again, the first 2ϱ term can be either eliminated or approximated by a constant, β_i^U, which should again be an upper bound for the sum it replaces. While it is not really germane to this discussion, it should be noted that β_i^U is significantly larger than β_i^L. As a result, estimates are inherently biased a little too high, especially in cases where there is significant node wait. This bias shows up both in simulation and in "real life", and will be of some concern in Chapter 6.

Let the p-th *inquiry pair* be inquiries $2p$ and $2p+1$. There will be q inquiry pairs, each consisting of a forward and a backward inquiry. Let $\mathcal{L}_i^p$ be the lower bound of the skew interval generated by the intersection of the intervals for N_i from the p-th inquiry pair, and $\mathcal{U}_i^p$ be the upper bound of the skew interval generated by the intersection of the intervals for N_i from the p-th inquiry pair. Given the distribution and density functions $F_{\underset{0\to i}{\mathcal{X}}}$, $f_{\underset{0\to i}{\mathcal{X}}}$, $F_{\underset{i\to 0}{\mathcal{X}}}$, $f_{\underset{i\to 0}{\mathcal{X}}}$, $F_{\underset{i\to k}{\mathcal{X}}}$, $f_{\underset{i\to k}{\mathcal{X}}}$, $F_{\underset{k\to i}{\mathcal{X}}}$, and $f_{\underset{k\to i}{\mathcal{X}}}$ of $\underset{0\to i}{\mathcal{X}}$, $\underset{i\to 0}{\mathcal{X}}$, $\underset{i\to k}{\mathcal{X}}$, and $\underset{k\to i}{\mathcal{X}}$, the distribution and density functions of $\mathcal{L}_i^p$ and $\mathcal{U}_i^p$ are given in Equations (5.17) through (5.20). The constant α_{il0} has been left out of these equations, as it will be taken care of later.

What is wanted is the maximum of the $\mathcal{L}_i^p$'s and the minimum of the $\mathcal{U}_i^p$'s. So two more random variables, MAX_i^{2q} and MIN_i^{2q}, are defined for the maximum lower bound and the minimum upper bound.

$$MAX_i^{2q} = \max\{\mathcal{L}_i^p : 0 \leq p < q\}$$

$$F_{\mathcal{L}_i^p}(x) = \left(1 - F_{\underset{i\to k}{\mathcal{X}}}\left(\frac{-x - 2\varrho\beta_i^L - 2\varrho\lambda(2q-2p-1)}{1+\varrho}\right)\right) \times \left(1 - F_{\underset{i\to 0}{\mathcal{X}}}\left(\frac{-x - 2\varrho\beta_i^L - 2\varrho\lambda(2q-2p-2)}{1+\varrho}\right)\right) \tag{5.17}$$

$$\begin{aligned} f_{\mathcal{L}_i^p}(x) = {} & f_{\underset{i\to k}{\mathcal{X}}}\left(\frac{-x - 2\varrho\beta_i^L - 2\varrho\lambda(2q-2p-1)}{1+\varrho}\right) \\ & \times \left(1 - F_{\underset{i\to 0}{\mathcal{X}}}\left(\frac{-x - 2\varrho\beta_i^L - 2\varrho\lambda(2q-2p-2)}{1+\varrho}\right)\right)\frac{1}{1+\varrho} \\ & + \left(1 - F_{\underset{i\to k}{\mathcal{X}}}\left(\frac{-x - 2\varrho\beta_i^L - 2\varrho\lambda(2q-2p-1)}{1+\varrho}\right)\right) \\ & \times f_{\underset{i\to 0}{\mathcal{X}}}\left(\frac{-x - 2\varrho\beta_i^L - 2\varrho\lambda(2q-2p-2)}{1+\varrho}\right)\frac{1}{1+\varrho} \end{aligned} \tag{5.18}$$

$$F_{\mathcal{U}_i^p}(x) = 1 - \left(1 - F_{\underset{0\to i}{\mathcal{X}}}\left(\frac{x - 2\varrho\beta_i^U - 2\varrho\lambda(2q-2p-1)}{1+\varrho}\right)\right) \times \left(1 - F_{\underset{k\to i}{\mathcal{X}}}\left(\frac{x - 2\varrho\beta_i^U - 2\varrho\lambda(2q-2p-2)}{1+\varrho}\right)\right) \tag{5.19}$$

$$\begin{aligned} f_{\mathcal{U}_i^p}(x) = {} & f_{\underset{0\to i}{\mathcal{X}}}\left(\frac{x - 2\varrho\beta_i^U - 2\varrho\lambda(2q-2p-1)}{1+\varrho}\right) \\ & \times \left(1 - F_{\underset{k\to i}{\mathcal{X}}}\left(\frac{x - 2\varrho\beta_i^U - 2\varrho\lambda(2q-2p-2)}{1+\varrho}\right)\right)\frac{1}{1+\varrho} \\ & + \left(1 - F_{\underset{0\to i}{\mathcal{X}}}\left(\frac{x - 2\varrho\beta_i^U - 2\varrho\lambda(2q-2p-1)}{1+\varrho}\right)\right) \\ & \times f_{\underset{i\to k}{\mathcal{X}}}\left(\frac{x - 2\varrho\beta_i^U - 2\varrho\lambda(2q-2p-2)}{1+\varrho}\right)\frac{1}{1+\varrho} \end{aligned} \tag{5.20}$$

$$MIN_i^{2q} = \min\{\mathcal{U}_i^p : 0 \le p < q\}$$

The distribution functions for MAX_i^{2q} and MIN_i^{2q} can be expressed in terms of the distribution functions of $\mathcal{L}_i^p$ and $\mathcal{U}_i^p$:

$$F_{MAX_i^{2q}}(x) = \prod_{p=0}^{q-1} F_{\mathcal{L}_i^p}(x) \tag{5.21}$$

$$f_{MAX_i^{2q}}(x) = \sum_{j=0}^{q-1} \left(\left(\prod_{p=0}^{j-1} F_{\mathcal{L}_i^p}(x) \right) f_{\mathcal{L}_i^j}(x) \left(\prod_{p=j+1}^{q-1} F_{\mathcal{L}_i^p}(x) \right) \right) \tag{5.22}$$

$$F_{MIN_i^{2q}}(x) = 1 - \prod_{p=0}^{q-1} (1 - F_{\mathcal{U}_i^p}(x)) \tag{5.23}$$

$$f_{MIN_i^{2q}}(x) = \sum_{j=0}^{q-1} \left(\left(\prod_{p=0}^{j-1} (1 - F_{\mathcal{U}_i^p}(x)) \right) f_{\mathcal{U}_i^j}(x) \left(\prod_{p=j+1}^{q-1} (1 - F_{\mathcal{U}_i^p}(x)) \right) \right) \tag{5.24}$$

Finally, it is possible to determine the distribution of Ω_i^{2q}, the width of the intersection interval for N_i. At this point, the α_{i0}'s that were ignored earlier must be taken into account.

$$\begin{aligned} F_{\Omega_i^{2q}}(x) &= P[(MIN_i^{2q} + \alpha_{i0}) - (MAX_i^{2q} + \alpha_{i0}) < x] \\ &= P[MIN_i^{2q} - MAX_i^{2q} < x]. \end{aligned} \tag{5.25}$$

Random variable MIN_i^{2q} depends on the values of $\mathcal{U}_i^p$, $0 \le p < q$. Each $\mathcal{U}_i^p$ depends on the values of $\mathcal{X}_{0\to i}^{2p}$, and $\mathcal{X}_{k\to i}^{2p+1}$. Similarly, MAX_i^{2q} ultimately depends on the values of $\mathcal{X}_{i\to k}^{h}$ and $\mathcal{X}_{i\to 0}^{h}$. Therefore, MIN_i^{2q} and MAX_i^{2q} depend on different random variables. And, since each $\mathcal{X}_{i\to j}^{h}$ is assumed to be independent of the others, MIN_i^{2q} and MAX_i^{2q} are also independent. The distribution of Ω_i^{2q} can therefore be computed by a simple convolution integral:

$$F_{\Omega_i^{2q}}(x) = \int_{-\infty}^{\infty} dy \int_{-\infty}^{x+y} f_{MAX_i^{2q}}(y) f_{MIN_i^{2q}}(z) dz. \tag{5.26}$$

Since $f_{MAX_i^{2q}}(y)$ does not depend on z, it can be moved outside the second integral. The integral $\int_{-\infty}^{x+y} f_{MIN_i^{2q}}(z)dz$ evaluates to $F_{MIN_i^{2q}}(x+y) - F_{MIN_i^{2q}}(-\infty)$. Since MIN_i^{2q} will always be greater than 0, $F_{MIN_i^{2q}}(x) = 0$ when $x < 0$, so Equation (5.26) reduces to:

$$F_{\Omega_i^{2q}}(x) = \int_{-\infty}^{\infty} f_{MAX_i^{2q}}(y) F_{MIN_i^{2q}}(x+y) dy$$

The range on integral of Equation (5.2.3) can be further reduced by noting that MAX_i^{2q} must be less than 0 and MIN_i^{2q} must be greater than 0, so $f_{MAX_i^{2q}}(y) = 0$ when $y > 0$ and $f_{MIN_i^{2q}}(x+y) = 0$ when $x + y < 0$. The final result is:

$$F_{\Omega_i^{2q}}(x) = \int_{-x}^{0} f_{MAX_i^{2q}}(y) F_{MIN_i^{2q}}(x+y)dy \tag{5.27}$$

5.2.4 Examples with $\varrho = 0$

Provided the distributions of the various $\underset{i\to j}{\mathcal{X}}$'s are known, Equation (5.27) gives a reasonable, perhaps slightly pessimistic, characterization of the width of N_i's intersection interval. Unfortunately, while the expression in Equation (5.27) is simple, calculation may not be. Equations (5.22) and (5.24), along with Equations (5.17) through (5.20), show $f_{MAX_i^{2q}}$ and $f_{MIN_i^{2q}}$ to be complicated sums of products, with the number of terms growing rapidly with q. This makes $F_{\Omega_i^{2q}}(x)$ difficult to evaluate, especially for large q.

For a first approximation, one can assume that the effects of clock drift during the estimation process are negligible. This is normally not too bad an assumption, ϱ often has a value in the neighborhood of 10^{-5} and is multiplied by β_i^L, β_i^U, and λ, whose values are usually approximately the time required for an inquiry. Since the time required for an inquiry is normally only one or two orders of magnitude greater than the desired interval width, the effects of ϱ terms are small in comparison.

Removing ϱ greatly simplifies Equations (5.17) through (5.20), resulting in new functions $F_{\mathcal{L}_i}$, $f_{\mathcal{L}_i}$, $F_{\mathcal{U}_i}$, and $f_{\mathcal{U}_i}$, which are independent of p. As a result, Equations (5.21) through (5.24) are also simplified, having the following form:

$$F_{MAX_i^{2q}}(x) = \left(F_{\mathcal{L}_i}(x)\right)^q \tag{5.28}$$

$$f_{MAX_i^{2q}}(x) = q\left(F_{\mathcal{L}_i}(x)\right)^{q-1} f_{\mathcal{L}_i}(x) \tag{5.29}$$

$$F_{MIN_i^{2q}}(x) = 1 - \left(1 - F_{\mathcal{U}_i}(x)\right)^q \tag{5.30}$$

$$f_{MIN_i^{2q}}(x) = q\left(1 - F_{\mathcal{U}_i}(x)\right)^{q-1} f_{\mathcal{U}_i}(x) \tag{5.31}$$

The remainder of this section will use these equations with different distributions for $\underset{i\to j}{\mathcal{X}}$ to estimate the number of inquiries needed for various values of k and the maximum allowable uncertainty ε.

Normally-Distributed Delays

It is commonly assumed that the delay for each "hop" taken by a synchronization message is independent of the delays for any other hops, and that the distributions of the

delays are identical for each hop. This is not a bad assumption in a homogeneous distributed system, such as a hypercube, where all nodes and communications links are identical. For this example things will be simplified a little more by assuming that all synchronization messages are sent only one "hop". Then $\underset{i\to i+i}{d}$ is a constant that does not depend on i, and the distributions for $\underset{i\to i+1}{\mathcal{X}}$ and $\underset{i+1\to i}{\mathcal{X}}$ are identical, and also do not depend on i, i.e., for $0 \le i, j < k$ and all x:

$$F_{\underset{i\to i+1}{\mathcal{X}}}(x) = F_{\underset{i+1\to i}{\mathcal{X}}}(x) = F_{\underset{j\to j+1}{\mathcal{X}}}(x) = F_{\underset{j+1\to j}{\mathcal{X}}}(x)$$

So $\underset{i\to j}{\mathcal{X}^h}$ is actually the sum of a number of independent, identically distributed random variables. The central limit theorem states that the distribution of a sum of a number of mutually independent identically distributed random variables with finite variances can be approximated by a normal distribution. In particular, the density function for $\underset{i\to j}{\mathcal{X}}$ will be the following:

$$f_{\underset{i\to j}{\mathcal{X}}}(x) = \frac{1}{\sqrt{2\pi|i-j|\sigma^2}} e^{-\frac{1}{2}\frac{(x-|i-j|)^2\mu^2}{|i-j|}\sigma^2} \tag{5.32}$$

Where μ is the mean value of $\underset{i\to i+1}{\mathcal{X}}$, and σ is its standard deviation. The distribution function can be expressed in terms of the error function, $Erf(x) = (2/\sqrt{\pi})\int_0^x e^{-t^2}dt$,

$$F_{\underset{i\to j}{\mathcal{X}}}(x) = \frac{1}{2} + \frac{1}{2} Erf\left(\frac{x-|j-i|\mu}{\sqrt{2|j-i|}\,\sigma}\right) \tag{5.33}$$

Substitution into Equations (5.17) through (5.20), and from there into Equations (5.28) through (5.31) is straightforward. Substitution into Equation (5.27), however, produces a complex and difficult to integrate function. Mathematica [46] and Maple V [6, 7, 8] were used to evaluate the integral for various values of i, k, q, and x. To check the results of the analysis, a simple simulation was done. The simulator simply chose values of $\underset{j\to j+1}{x}$ randomly using a normal distribution with mean μ and standard deviation σ.

The values of $\underset{j\to j+1}{d}$, μ, and σused by other researchers are used here in order to facilitate comparisons. Both Cristian [11] and Arvind [3] use $\underset{j\to j+1}{d} = 2.11$msec., and $\mu = 0.34$ msec. Cristian does not give a value for σ, but Arvind gives $\sigma = 1$. To see the effects of variance in delay, $\sigma = 0.3$ is also considered.

Figure 5.4 plots $f_{\mathcal{L}_i^p}(x)$ and $f_{\mathcal{U}_i^p}(x)$ for both $i = 8$ and $i = 2$. Since ϱ is assumed 0, the graphs do not depend on p and q, and thus are the same for all inquiry pairs. The graphs show that the interval for N_2 will almost certainly be smaller than the interval for N_8, as

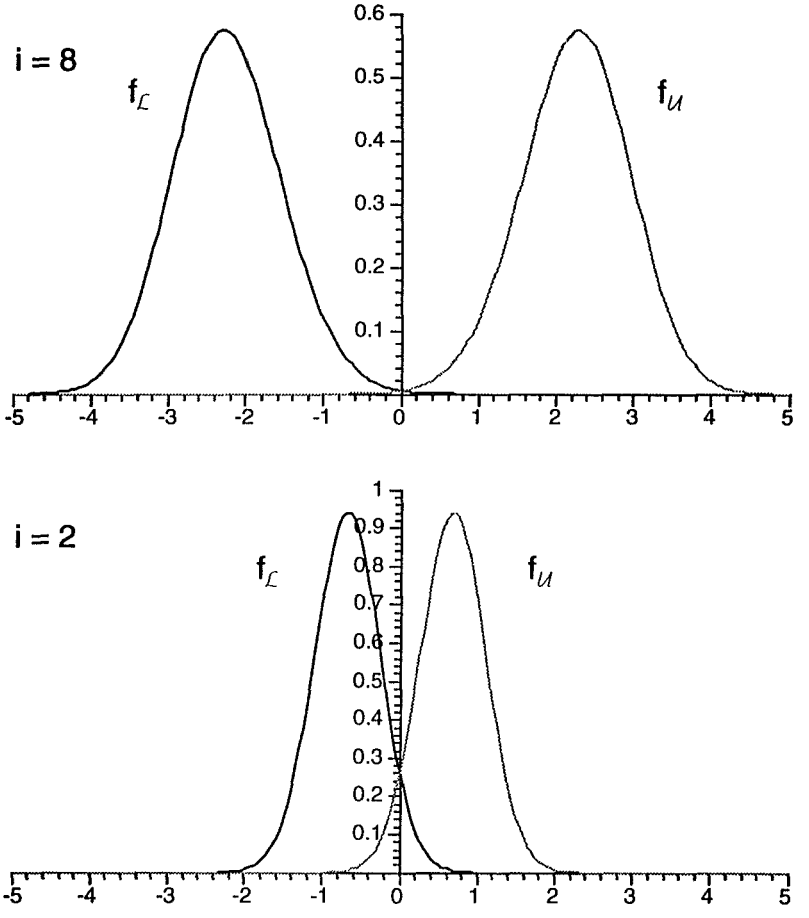

Figure 5.4: $f_{\mathcal{L}_i^p}(x)$ and $f_{\mathcal{U}_i^p}(x)$, assuming $\alpha_{i0} = 0$, $k = 16$, $i = 8$ and $i = 2$, and normally distributed delays with $\mu = 0.34$ and $\sigma = 0.3$

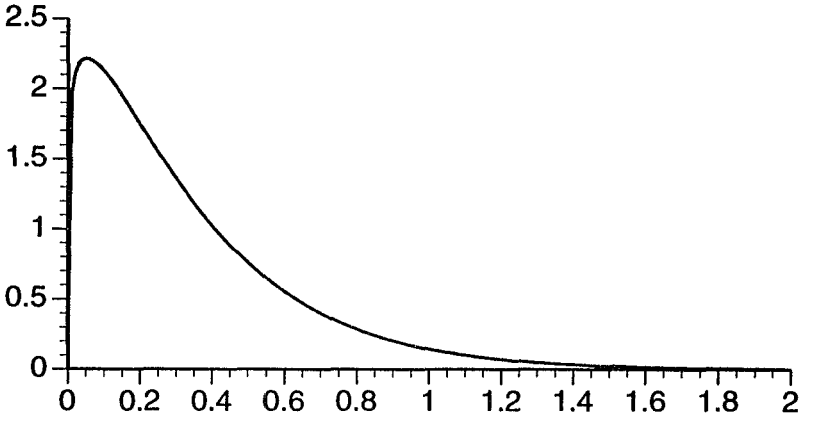

Figure 5.5: Weibull distribution with $\mu = 0.34$ and $\sigma = 0.3$

was discussed in Section 5.2.2. The graphs also show a problem with assuming a normal distribution. Since normally-distributed random variables have no absolute lower bound, the normally-distributed delays will sometimes be less than $_{i\rightarrow i+1}d$, which the estimation algorithm assumes cannot happen. The results can be seen in Figure 5.4 where the distributions cross the y-axis, allowing positive lower bounds and negative upper bounds. Since $\alpha_{i|0} = 0$, such intervals will not contain the actual skew value, it may even be that the lower bound is greater than the upper bound. The error introduced is small enough to be ignored, at least for i near the "middle", and $\sigma = 0.3$. However, for other nodes, and larger values of σ, one should keep this source of error in mind.

The simulator also ran into this problem, and sometimes chose negative values of $_{i\rightarrow i+1}x$, which often led to intervals whose lower bound was greater than their upper bound. The simulator ignored these situations, since forcing $_{i\rightarrow i+1}x \geq 0$ would alter the delay distribution and simulation results would no longer match analytical results. However, the simulator was also run using a Weibull distribution for the delays. The Weibull distribution has a definite minimum, and it is relatively simple to generate Weibully distributed random numbers. However, the Weibull distribution does not lend itself to easy analysis. Figure 5.5 shows a Weibull density function with $\mu = 0.34$ and $\sigma = 0.3$.

The results of the analysis and simulation are given in Tables 5.1 through 5.4. One analytical and six simulation results are presented. The single analytical result is the value of $F_{\Omega_i^{2q}}(\varepsilon)$, calculated with Equation (5.2.3)[2], assuming normally-distributed delays. The simulation results consist of the following three values for both the normal and Weibull distributions:

P_i: The probability that the width of N_0's intersection interval for N_i is less than the specified value.

$P_{0|i}$: The probability that the width of *all* of N_0's intersection intervals are less than 2ε, given that the width of N_0's intersection interval for N_i is less than the specified value. This is an indication of the dependency between the width of N_0's interval for N_i and the width of the rest of N_0's intervals.

$P_{A|i}$: The probability that the width of *all* intersection intervals at *all* k nodes are less than 2ε, given that the width of N_0's intersection interval for N_i is less than the specified

[2]Equation (5.2.3) is used instead of Equation (5.27) since the lack of a lower bound on the normal distribution means that $MAX_i^{2q} \leq 0$ and $MIN_i^{2q} \geq 0$ no longer holds.

				normal				*Weibull*			
k	i	$2q$	$F_{\Omega_i^{2q}}(2)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}
16	8	78	0.90690	0.9099	0.9472	0.3636	1.0000	0.2767	0.6757	0.0015	1.0000
16	7	78	0.95270	0.9544	0.9029	0.3465	1.0000	0.5530	0.3374	0.0007	1.0000
16	5	78	0.99998	1.0000	0.8618	0.3306	1.0000	1.0000	0.1874	0.0004	1.0000
16	8	194	0.99906	0.9991	0.9998	0.9850	1.0000	0.6964	0.9381	0.0668	1.0000
16	7	194	0.99985	0.9999	0.9990	0.9843	1.0000	0.9312	0.7021	0.0498	1.0000
16	6	194	1.00000	1.0000	0.9989	0.9843	1.0000	0.9999	0.6539	0.0464	1.0000
16	8	296	0.99999	1.0000	1.0000	0.9998	1.0000	0.8737	0.9873	0.2889	1.0000
32	16	15038	0.90003	-	-	-	-	-	-	-	-
32	15	15038	0.94768	-	-	-	-	-	-	-	-
32	13	15038	0.99997	-	-	-	-	-	-	-	-
32	16	38288	0.99900	-	-	-	-	-	-	-	-
32	15	38288	0.99983	-	-	-	-	-	-	-	-
32	14	38288	1.00000	-	-	-	-	-	-	-	-
32	16	58718	0.99999	-	-	-	-	-	-	-	-

Table 5.1: Probability of convergence when $\varepsilon = 1.0$, assuming normally distributed delays with $\underset{i \to i+1}{d} = 2.11$, $\sigma = 0.3$, and $\mu = 0.34$.

value. This is an indication of the dependency between the width of N_0's interval for N_i and the width of all intervals at all nodes.

P_{Av}: The probability that the average width of *all* intersection intervals at *all* k nodes is less than 2ε. This probability is not conditioned on any event. This is the probability that all nodes will be able to use the adjustment algorithm of Chapter 3 with all estimates.

Each run of the simulator consisted of multiple "trials". A trial consisted of performing the specified number of inquiries, and noting which intervals converged. For larger values of k and q the cost of simulation became prohibitive. Weeks or months of computer time were required for a single run. In some cases the cost of simulation was deemed too high and results are not available.

The high cost of simulation also makes it difficult to determine the standard deviation of the results. With single runs so expensive it is not feasible to make the multiple runs needed to compute the standard deviation of the results. Multiple runs were done for some

				normal				*Weibull*			
k	i	$2q$	$F_{\Omega_i^{2q}}(2)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}
16	8	6	0.93306	0.9333	0.9006	0.4628	0.9589	0.8790	0.8980	0.4082	0.9933
16	7	6	0.93546	0.9360	0.8983	0.4617	0.9589	0.8917	0.8855	0.4023	0.9933
16	2	6	0.99336	0.9935	0.8456	0.4344	0.9589	0.9991	0.7899	0.3588	0.9933
16	8	14	0.99938	0.9994	0.9983	0.9772	1.0000	0.9980	0.9978	0.9637	1.0000
16	7	14	0.99943	0.9994	0.9982	0.9771	1.0000	0.9985	0.9973	0.9629	1.0000
16	3	14	0.99997	1.0000	0.9977	0.9765	1.0000	1.0000	0.9958	0.9617	1.0000
16	8	22	1.00000	1.0000	1.0000	0.9998	1.0000	1.0000	1.0000	0.9994	1.0000
32	16	12	0.90983	0.9105	0.8231	0.2220	0.9752	0.0994	0.2922	0.0001	0.6546
32	15	12	0.91110	0.9122	0.8222	0.2217	0.9752	0.1113	0.2623	0.0001	0.6546
32	2	12	0.99993	0.9999	0.7494	0.2019	0.9752	1.0000	0.0290	0.0000	0.6546
32	16	30	0.99921	0.9992	0.9968	0.9429	1.0000	0.3745	0.5257	0.0022	1.0000
32	15	30	0.99924	0.9993	0.9969	0.9430	1.0000	0.4055	0.4848	0.0020	1.0000
32	8	30	0.99996	1.0000	0.9962	0.9425	1.0000	0.9989	0.1967	0.0008	1.0000
32	16	46	0.99999	1.0000	1.0000	0.9988	1.0000	0.5871	0.6484	0.0119	1.0000
64	32	40	0.90029	0.9021	0.7678	0.0879	0.9998	0.0000	0.0000	0.0000	0.0000
64	31	40	0.90109	0.9025	0.7672	0.0879	0.9998	0.0000	0.0000	0.0000	0.0000
64	9	40	0.99999	1.0000	0.6921	0.0796	0.9998	0.9988	0.0000	0.0000	0.0000
64	32	102	0.99908	0.9991	0.9956	0.8857	1.0000	0.0000	0.0000	0.0000	0.0000
64	31	102	0.99910	0.9991	0.9956	0.8856	1.0000	0.0000	0.0000	0.0000	0.0000
64	20	102	0.99998	1.0000	0.9946	0.8847	1.0000	0.0738	0.0000	0.0000	0.0000
64	32	160	0.99999	1.0000	0.9999	0.9974	1.0000	0.0000	0.0000	0.0000	0.0038

Table 5.2: Probability of convergence when $\varepsilon = 1.0$, assuming normally distributed delays with $\underset{i \to i+1}{d} = 2.11$, $\sigma = 1.0$, and $\mu = 0.34$.

				normal				*Weibull*			
k	i	$2q$	$F_{\Omega_i^{2q}}(4)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}
16	8	8	0.91595	0.9178	0.9503	0.5125	1.0000	0.9230	0.9585	0.5349	1.0000
16	7	8	0.94685	0.9480	0.9200	0.4962	1.0000	0.9568	0.9246	0.5160	1.0000
16	4	8	1.00000	1.0000	0.8722	0.4704	1.0000	1.0000	0.8844	0.4940	1.0000
16	8	20	0.99936	0.9994	0.9997	0.9896	1.0000	0.9996	0.9999	0.9941	1.0000
16	7	20	0.99981	0.9998	0.9993	0.9891	1.0000	0.9999	0.9995	0.9922	1.0000
16	6	20	1.00000	1.0000	0.9991	0.9890	1.0000	1.0000	0.9994	0.9921	1.0000
16	8	30	0.99999	1.0000	1.0000	0.9998	1.0000	1.0000	1.0000	0.9999	1.0000
32	16	786	0.90064	0.9063	0.9385	0.1593	1.0000	0.0258	0.2674	0.0000	1.0000
32	15	786	0.94244	0.9459	0.8992	0.1530	1.0000	0.0716	0.0965	0.0000	1.0000
32	12	786	1.00000	1.0000	0.8509	0.1444	1.0000	0.9973	0.0071	0.0000	1.0000
32	16	1992	0.99900	0.999	1.000	0.969	1.000	0.117	0.488	0.000	1.000
32	15	1992	0.99977	1.000	0.999	0.968	1.000	0.276	0.210	0.000	1.000
32	14	1992	1.00000	1.000	0.999	0.968	1.000	0.767	0.075	0.000	1.000
32	16	3058	0.99999	1.000	1.000	1.000	1.000	0.219	0.614	0.000	1.000

Table 5.3: Probability of convergence when $\varepsilon = 2$, assuming normally distributed delays with ${}_{i\rightarrow i+1}d = 2.11$, $\sigma = 0.3$, and $\mu = 0.34$.

small values of k and q. Comparison of these runs shows that results from runs containing a million or more trials are accurate to well within ± 0.0005, and runs containing one hundred thousand or more trials are accurate to within ± 0.005. For this reason, results from runs of one million or more trials are reported to 4 decimal places, while results from runs of between one hundred thousand and one million trials are reported to 3 decimal places.

One would expect that since the Weibull distribution does not produce negative values of ${}_{i\rightarrow i+1}\mathcal{X}$, it would be slower to converge than the normal distribution. This does not turn out to be true. The Weibull distribution actually converges faster than the normal distribution in some cases. This is especially true when k is small while σ and ε are large.

Tables 5.2 and 5.4 show best how the difference between the normal and Weibull distribution varies. In both tables there is little difference between the two distributions when $k = 16$, considerable difference when $k = 32$, and the Weibull distribution yields probabilities near 0 when $k = 64$. And, while for both distributions the probabilities increase as the difference between i and $k/2$ increases, the probabilities for the Weibull distribution increase faster. But the most obvious difference between the two tables is that the probabilities for

				normal				*Weibull*			
k	i	$2q$	$F_{\Omega_i^{2q}}(4)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}
16	8	4	0.94692	0.9472	0.9386	0.5957	0.9877	0.9824	0.9808	0.8424	0.9990
16	7	4	0.94950	0.9497	0.9362	0.5941	0.9877	0.9839	0.9794	0.8413	0.9990
16	2	4	0.99852	0.9985	0.8903	0.5650	0.9877	0.9995	0.9642	0.8282	0.9990
16	8	10	0.99982	0.9998	0.9996	0.9939	1.0000	1.0000	1.0000	0.9999	1.0000
16	7	10	0.99984	0.9998	0.9996	0.9939	1.0000	1.0000	1.0000	0.9999	1.0000
16	2	10	0.99999	1.0000	0.9994	0.9937	1.0000	1.0000	1.0000	0.9998	1.0000
16	8	14	0.99999	1.0000	1.0000	0.9998	1.0000	1.0000	1.0000	1.0000	1.0000
32	16	8	0.91122	0.9119	0.8566	0.2918	0.9866	0.6848	0.7472	0.0865	0.9976
32	15	8	0.91264	0.9134	0.8553	0.2908	0.9866	0.6971	0.7341	0.0853	0.9976
32	3	8	0.99988	0.9999	0.7812	0.2656	0.9866	1.0000	0.5115	0.0591	0.9976
32	16	20	0.99926	0.9993	0.9975	0.9547	1.0000	0.9795	0.9758	0.6735	1.0000
32	15	20	0.99929	0.9993	0.9975	0.9547	1.0000	0.9818	0.9735	0.6714	1.0000
32	8	20	0.99998	1.0000	0.9968	0.9539	1.0000	1.0000	0.9557	0.6539	1.0000
32	16	30	0.99999	1.0000	1.0000	0.9987	1.0000	0.9983	0.9976	0.9420	1.0000
64	32	28	0.90512	0.9070	0.7969	0.1228	0.9999	0.0027	0.0739	0.0000	0.6124
64	31	28	0.90592	0.9074	0.7963	0.1227	0.9999	0.0029	0.0599	0.0000	0.6124
64	10	28	0.99995	0.9999	0.7226	0.1114	0.9999	1.0000	0.0002	0.0000	0.6124
64	32	70	0.99907	0.9991	0.9939	0.8971	1.0000	0.0154	0.1088	0.0000	1.0000
64	31	70	0.99909	0.9991	0.9959	0.8972	1.0000	0.0172	0.0930	0.0000	1.0000
64	19	70	0.99999	1.0000	0.9950	0.8962	1.0000	0.9542	0.0017	0.0000	1.0000
64	32	110	0.99999	1.0000	0.9999	0.9977	1.0000	0.0345	0.1520	0.0000	1.0000

Table 5.4: Probability of convergence when $\varepsilon = 2$, assuming normally distributed delays with $\underset{i \to i+1}{d} = 2.11$, $\sigma = 1.0$, and $\mu = 0.34$.

the Weibull distribution are consistently higher in Table 5.4.

Since the only difference in parameters between Tables 5.2 and 5.4 is an increase in ε, it is reasonable to assume that this increase is the cause of the differences in the probabilities for the Weibull distribution. The likely explanation is that in the Weibull equivalent of Figure 5.4, the peaks are slightly further apart, but much narrower. As q increases, the width of the peaks for both distributions decreases, though the peaks for the Weibull distribution are always much narrower, and all peaks move towards 0, with the normal peaks closer to 0 than the Weibull peaks. However, while the peaks for the two distributions are getting both narrower and closer together, they are getting narrower faster than they are getting closer. So, when $\varepsilon = 2$, q is small enough that there is considerable overlap between the peaks of both distributions, and any width that contains most of the normal peaks will contain much of the Weibull peaks as well. But when $\varepsilon = 1$, q has increased to the point where there is little overlap between the peaks, and a width that encompasses most of the normal peaks will contain little of the Weibull peaks.

The narrower peaks generated by the Weibull distribution also explains why its probabilities increase faster as $|i - k/2|$ increases. As Figure 5.4 demonstrates, as i gets further away from $k/2$, the peaks move closer to 0. Because the peaks for the Weibull distribution are much narrower, bringing the peaks closer to 0 can greatly increase the percentage of the peaks within any specified width.

Comparison of the analytical and simulation results for the normal distribution shows a close match. Simulation and analytical results almost always match out to two decimal places, and are quite close out to three decimal places. This close agreement lends confidence to the values of $P_{0|i}$ and $P_{A|i}$ found by simulation. These results, in turn, show the dependency of the widths of the various intervals. For example, consider the first two rows of Table 5.2. Assume that N_0's interval for N_8 has converged, then $P_{0|8}$ should be no greater than the probability that the intervals for both N_7 and N_9 have converged. If intervals converged independently then the probability that the intervals for both N_7 and N_9 have converged is $0.9360^2 = 0.8761$, which is already less than the value determined through simulation. A similar relationship exists between $P_{0|i}$ and $P_{A|i}$. Assume that at each node the interval for the "middle" node has converged. The probability that all intervals have converged at any given node is then 0.9006. If independence is assumed between nodes, the probability that all intervals have converged at all nodes is then $0.9006^{16} = 0.1873$, significantly lower than the value produced by the simulation.

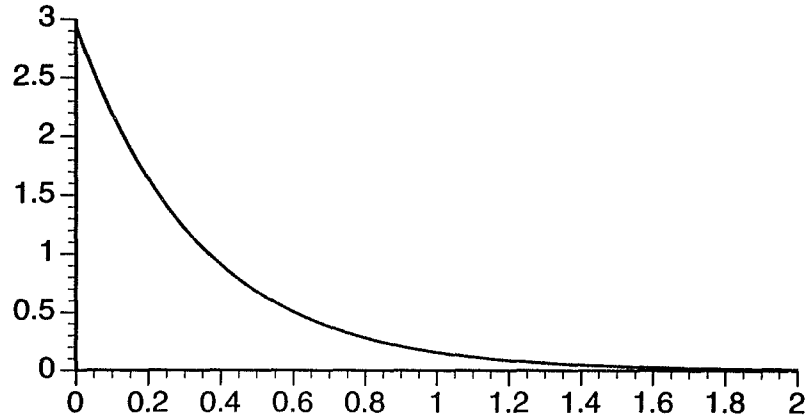

Figure 5.6: Exponential distribution with $\mu = 0.34$

A pleasant surprise is the value of P_{Av}, which consistently reaches 1.0 long before the probability of convergence of $N_{k/2}$ reaches 0.999. This means if the adjustment algorithm of Chapter 3 is used, the number of inquiries can be reduced considerably. The value of P_{Av} for the Weibull distribution generally matches that of the normal distribution, even when the probability of convergence for the Weibull distribution is much lower. Apparently, the slow convergence of nodes near $N_{k/2}$ is made up for by faster convergence of nodes near N_1 and N_{k-1}.

Exponential Delays

As the examples of the previous section show, the lack of a lower bound means a normal distribution is not a good approximation when σ is much larger than μ. The advantage of a normal distribution is the ease with which one may determine the distribution of the sum of a number of independent normally distributed random variables. Another distribution may be used instead, if the distribution of the sum is known. The exponential distribution has this property, one may easily derive the distribution of the sum of independent identically distributed exponential random variables. As can be seen in Figure 5.6, the exponential distribution also has a definite minimum, avoiding the problems encountered with the normal distribution. Finally, network delays are often assumed to be exponentially distributed. The primary disadvantage of the exponential distribution is that μ and σ cannot be assigned independently. In particular, $\sigma = \mu$.

The density function for a single hop using exponential delays is:

$$f_{\underset{i \to i+1}{\mathcal{X}}}(x) = f_{\underset{i+1 \to i}{\mathcal{X}}}(x) = \frac{1}{\mu} e^{-x/\mu} \tag{5.34}$$

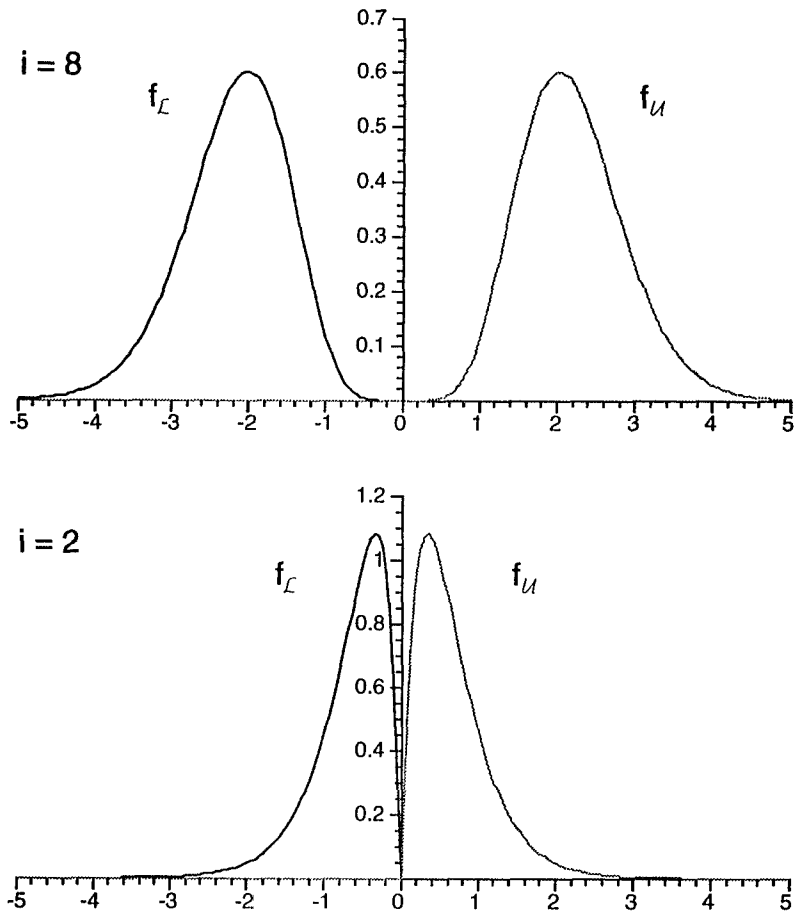

Figure 5.7: $f_{\mathcal{L}_i^p}(x)$ and $f_{\mathcal{U}_i^p}(x)$, assuming $\alpha_{i0} = 0$, $k = 16$, $i = 8$ and $i = 2$, and exponentially distributed delays with $\mu = 0.34$

The density function for multiple hops is easily derived via repeated convolution:

$$f_{\underset{i \to j}{\mathcal{X}}}(x) = \frac{x^{(|i-j|-1)} e^{(-x/\mu)}}{(|i-j|-1)! \mu^{(|i-j|)}} \tag{5.35}$$

For purposes of comparison, Figure 5.7 plots $f_{\mathcal{L}_i^p}(x)$ and $f_{\mathcal{U}_i^p}(x)$ when the distribution of $\underset{i \to i+1}{\mathcal{X}}$ is assumed to be exponential. Note that, in contrast to the normally-distributed delays, the lower bound is always negative, while the upper bound is always positive, and as a result the lower bound of the intervals is always less than the upper bound.

Results are presented in Table 5.5. In this case, simulation results for only the exponential distribution are presented.

Unfortunately, while Equation (5.35) is the distribution for the sum of an arbitrary number of independent identically distributed exponential random variables, it is not particularly simple, especially in comparison to the normal distribution. In particular, both

				exponential			
k	i	$2q$	$F_{\Omega_i^{2q}}(2)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}
16	8	156	0.90083	0.9066	0.9830	0.3862	1.0000
16	7	156	0.98354	0.9848	0.9048	0.3554	1.0000
16	6	156	0.99998	1.0000	0.8912	0.3501	1.0000
16	8	390	0.99901	0.9992	1.0000	0.9880	1.0000
16	7	390	0.99999	1.0000	0.9992	0.9872	1.0000
16	8	596	0.99999	1.0000	1.0000	0.9998	1.0000

(a) $\varepsilon = 1$

				exponential			
k	i	$2q$	$F_{\Omega_i^{2q}}(4)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}
16	8	8	0.95625	0.9573	0.9735	0.6698	1.0000
16	7	8	0.97395	0.9758	0.9584	0.6748	1.0000
16	5	8	0.99972	0.9997	0.9323	0.6417	1.0000
16	4	8	1.00000	1.0000	0.9321	0.6414	1.0000
16	8	16	0.99924	0.9993	0.9996	0.9877	1.0000
16	7	16	0.99975	0.9998	0.9992	0.9873	1.0000
16	6	16	0.99999	1.0000	0.9989	0.9870	1.0000
16	8	24	0.99999	1.0000	1.0000	0.9998	1.0000

(b) $\varepsilon = 2$

Table 5.5: Probability of convergence assuming exponentially distributed delays with $\underset{i \to i+1}{d} = 2.11$, and $\mu = 0.34$.

Mathematica and Maple had difficulty doing the integration with the exponential distribution. During the search for alternative methods it was discovered that Maple could generate C [19] code which allowed the function being integrated to be evaluated at any point. With this, it was possible to write a program to do the integration numerically. Accuracy of this method was checked by trying it with the normal distribution. The results matched those in Tables 5.1 through 5.4, and took far less time to produce. However, computational difficulties still made it impossible to obtain results for large k.

A comparison between the normal and exponential distributions yields conclusions similar to those for the comparison of the normal and Weibull distributions. Comparing Tables 5.1 and 5.6a shows much slower convergence for the exponential distribution when $\varepsilon = 1$. Comparing Tables 5.3 and 5.6b shows faster convergence for the exponential distribution when $k = 16$, with the normal distribution having faster convergence for larger k. Like the Weibull distribution, the exponential distribution generates narrow, high peaks, which move towards 0 more slowly than those of the normal distribution. A careful comparison of Figures 5.4 and 5.7 bears this out. This parallel between the Weibull and exponential distributions is not surprising, their density curves have a similar shape, as a comparison of Figures 5.5 and 5.6 shows.

The analytical and simulation results are as close for the exponential distribution as they are for the normal distribution, and the same observations with respect to $P_{0|i}$ and $P_{A|i}$ hold. Perhaps more obvious in this case is the slightly higher probabilities obtained through simulation. This trend can also be seen with the normal distribution, though in that case the difference is less marked. This may be due to biases in the random number generation routines. More likely it is a reflection of the various approximations made during the analysis. The net result of these approximations is a slightly conservative probability.

5.2.5 Examples with $\varrho > 0$

The examples of Section 5.2.4 all assume that ϱ is small enough that its effects can be ignored. This allows the use of Equations (5.28) through (5.31) instead of the far more complicated Equations (5.21) through (5.24). While this may save computational expense, it is likely that the time required to complete the necessary number of inquiries may be non-trivial. This section considers whether or not ϱ can be safely ignored.

A fairly conservative value of 10^{-5} is used for ϱ. And, since ϱ is assumed non-zero, a value must be specified for λ also. For these examples $\lambda = 100$msec., or ten inquiries are

			normal				*Weibull*			
k	$2q$	$F_{\Omega_i^{2q}}(2)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}
16	102	0.90217	0.9398	0.9662	0.4766	1.0000	0.2593	0.6703	0.0010	1.0000
16	1000	0.99566	1.0000	1.0000	0.9999	1.0000	0.499	0.879	0.010	1.000
32	1000	0.00152	0.0054	0.0544	0.0000	1.0000	0.000	0.000	0.000	0.000
64	1000	0.00000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

(a) $\sigma = 0.3$

			normal				*Weibull*			
k	$2q$	$F_{\Omega_i^{2q}}(4)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}
16	6	0.93229	0.9332	0.9002	0.4612	0.9585	0.8778	0.8970	0.4055	0.9932
16	14	0.99934	0.9993	0.9982	0.9765	1.0000	0.9979	0.9976	0.9614	1.0000
16	22	0.99999	1.0000	1.0000	0.9997	1.0000	1.0000	1.0000	0.9993	1.0000
32	12	0.90833	0.9099	0.8221	0.2205	0.9747	0.0968	0.2902	0.0002	0.6428
32	30	0.99912	0.9992	0.9967	0.9407	1.0000	0.3540	0.5101	0.0017	1.0000
32	46	0.99999	1.0000	1.0000	0.9986	1.0000	0.5492	0.6580	0.0084	1.0000
64	42	0.90926	0.9134	0.7861	0.1021	0.9999	0.0000	0.0000	0.0000	0.0000
64	106	0.99906	0.9993	0.9961	0.8963	1.0000	0.000	0.000	0.000	0.000
64	166	0.99999	1.0000	1.0000	0.9976	1.0000	0.000	0.000	0.000	0.000

(b) $\sigma = 1.0$

Table 5.6: Probability of convergence when $\varepsilon = 1$ and $i = k/2$, assuming normally distributed delays with $\underset{i \to i+1}{d} = 2.11$, $\mu = 0.34$, $\varrho = 10^{-5}$, and $\lambda = 100$.

			normal				*Weibull*			
k	$2q$	$F_{\Omega_i^{2q}}(2)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}
16	8	0.91167	0.9161	0.9492	0.5074	1.0000	0.9211	0.9574	0.5282	1.0000
16	20	0.99912	0.9994	0.9997	0.9896	1.0000	0.9995	0.9998	0.9913	1.0000
16	32	0.99999	1.0000	1.0000	0.9999	1.0000	1.0000	1.0000	0.9999	1.0000
32	2000	0.26556	0.568	0.697	0.003	1.000	0.001	0.086	0.000	1.000
64	2000	0.00000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

(a) $\sigma = 0.3$

			normal				*Weibull*			
k	$2q$	$F_{\Omega_i^{2q}}(4)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}
16	4	0.94650	0.9473	0.9388	0.5957	0.9877	0.9824	0.9808	0.8420	0.9990
16	10	0.99981	0.9998	0.9996	0.9939	1.0000	1.0000	1.0000	0.9999	1.0000
16	14	1.00000	1.0000	1.0000	0.9998	1.0000	1.0000	1.0000	1.0000	1.0000
32	8	0.91028	0.9119	0.8572	0.2914	0.9860	0.6822	0.7465	0.0856	0.9975
32	20	0.99920	0.9993	0.9975	0.9544	1.0000	0.9784	0.9747	0.6616	1.0000
32	30	0.99999	1.0000	1.0000	0.9988	1.0000	0.9980	0.9972	0.9354	1.0000
64	28	0.90219	0.9053	0.7950	0.1204	0.9999	0.0025	0.0764	0.0000	0.5841
64	72	0.99908	0.9992	0.9965	0.9054	1.0000	0.013	0.098	0.000	1.000
64	112	0.99999	1.0000	1.0000	0.9977	1.0000	0.027	0.137	0.000	1.000

(b) $\sigma = 1.0$

Table 5.7: Probability of convergence when $\varepsilon = 2$ and $i = k/2$, assuming normally distributed delays with $\underset{i \to i+1}{d} = 2.11$, $\mu = 0.34$, $\varrho = 10^{-5}$, and $\lambda = 100$.

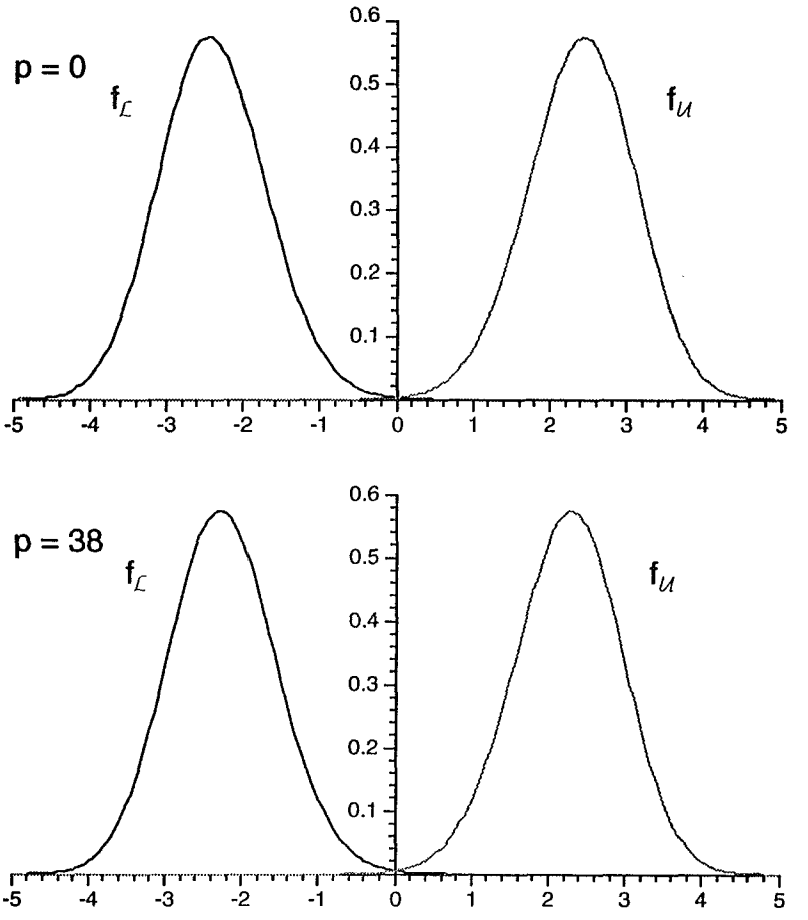

Figure 5.8: $f_{\mathcal{L}_i^p}(x)$ and $f_{\mathcal{U}_i^p}(x)$, assuming $\alpha_{i0} = 0$, $k = 16$, $i = 8$, $q = 39$, $p = 0$ and $p = 38$, $\varrho = 10^{-5}$, $\lambda = 100$, and normally distributed delays with $\mu = 0.34$, and $\sigma = 0.3$.

completed each second. Figure 5.8 gives the first indication that ϱ and λ can affect results. This figure plots $f_{\mathcal{L}_i^p}(x)$ and $f_{\mathcal{U}_i^p}(x)$ for N_8 for the first and last inquiry pairs when $q = 39$. Careful comparison of the two plots shows that the $f_{\mathcal{L}_i^p}(x)$ curve is further left when $p = 0$, and the $f_{\mathcal{U}_i^p}(x)$ curve is further right when $p = 0$. The result is that the interval generated by the first inquiry pair is likely to be wider than intervals generated by later inquiry pairs. The difference between curves is small enough that one might believe it to be insignificant, analysis shows otherwise. Tables 5.6 and 5.7 summarize the results.

A normal distribution is used for the analysis. Even though Section 5.2.4 exposed some difficulties with the normal distribution, it is the only available choice. The Weibull distribution does not allow analysis, and while the exponential distribution allows analysis, it does not allow alteration of σ, and it is computationally expensive to evaluate. Even using the normal distribution, neither Mathematica nor Maple were able to evaluate the

integral in Equation (5.2.3), and the alternative method used for exponential distributions had to be used. To reduce the computational load, results are only provided for $N_{k/2}$. The point about other nodes converging more quickly was made in Section 5.2.4, and holds here as well.

An immediate result of the choice of ϱ and λ is a bound on the value of $F_{\Omega_i^{2q}}(\varepsilon)$, independent of q. After a guess is made, the lower bound of its interval is decreased, and the upper bound of its interval is increased, at a rate of 2ϱ. If the actual skew is assumed to remain constant, both bounds of the interval will be at least 2ε from the actual skew value after a time of ε/ϱ. Such an interval is useless for convergence since it cannot be intersected with any other interval containing the actual skew and result in an intersection of width less than 2ε. When $\varepsilon = 1$msec., and $\varrho = 10^{-5}$, intervals reach this point after 100 seconds, which is the time taken for 1000 inquiries if $\lambda = 100$msec. When ε is increased to 2msec., the maximum "useful" number of inquiries increases to 2000. In cases where q reaches its maximum before the probability of convergence for $N_{k/2}$ reaches 0.99999, the probability of convergence when q is at its maximum is shown.

As the results of Tables 5.6 and 5.7 show, a non-zero ϱ can have considerable effect on the ability of the estimation algorithm to make accurate estimates. And, as pointed out above, the values of ε, ϱ, and λ place an effective limit on the value of q. But, as can also be seen from Tables 5.6 and 5.7, the effects of ϱ and λ are greatest in cases where q is large. There is little difference between corresponding entries of Tables 5.2 and 5.7b, nor between corresponding entries of Tables 5.4 and 5.8b. What differences do exist are found when $k = 64$, and involve an increase in q of no more than 3. The picture is very different when comparing Tables 5.1 and 5.7a and Tables 5.3 and 5.8a. Differences are found even when $k = 16$, and limits on q are reached for larger k.

Reducing q is the obvious method for reducing the effects of ϱ and λ. However, that may not be possible or practical. The value of q depends on k, μ, σ, and ε, of which only ε can usually be easily changed. But the value of ε is limited by the adjustment algorithm, and if it cannot be increased, q cannot be decreased. This leaves only the alternative of decreasing either ϱ, or λ, or both. The value of ϱ is determined by the hardware that generates the clock signal. Buying better quality hardware (if available and practical) can reduce ϱ. The value of λ is determined by limitations on network bandwidth, and by how much of it one is willing to let the clock distribution algorithm use. To get an idea of just how much can be gained from reductions in ϱ and λ, Tables 5.8 and 5.8 show the results when ϱ is reduced

			normal				*Weibull*			
k	$2q$	$F_{\Omega_i^{2q}}(2)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}
16	78	0.90616	0.9098	0.9473	0.3641	1.0000	0.2760	0.6751	0.0014	1.0000
16	194	0.99901	0.9991	0.9997	0.9845	1.0000	0.6948	0.9375	0.0660	1.0000
16	298	0.99999	1.000	1.000	1.000	1.000	0.874	0.987	0.288	1.000

(a) $\sigma = 0.3$

			normal				*Weibull*			
k	$2q$	$F_{\Omega_i^{2q}}(4)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}
16	6	0.93304	0.9334	0.9004	0.4620	0.9590	0.8792	0.8978	0.4077	0.9953
16	14	0.99938	0.9994	0.9983	0.9772	1.0000	0.9980	0.9978	0.9636	1.0000
16	22	1.00000	1.0000	1.0000	0.9997	1.0000	1.0000	1.0000	0.9994	1.0000
32	12	0.90980	0.9107	0.8231	0.2225	0.9753	0.0994	0.2919	0.0002	0.6541
32	30	0.99921	0.9992	0.9969	0.9432	1.0000	0.3748	0.5251	0.0022	1.0000
32	46	0.99999	1.0000	0.9999	0.9987	1.0000	0.5871	0.6845	0.0119	1.0000
64	40	0.90023	0.901	0.767	0.088	1.000	0.000	0.000	0.000	0.000
64	102	0.99908	0.999	0.996	0.885	1.000	0.000	0.000	0.000	0.000
64	160	0.99999	1.000	1.000	0.997	1.000	0.000	0.000	0.000	0.004

(b) $\sigma = 1.0$

Table 5.8: Probability of convergence when $\varepsilon = 1$ and $i = k/2$, assuming normally distributed delays with $\underset{i \to i+1}{d} = 2.11$, $\mu = 0.34$, $\varrho = 10^{-6}$, and $\lambda = 10$.

			normal				*Weibull*			
k	$2q$	$F_{\Omega_i^{2q}}(2)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}
16	8	0.91583	0.9179	0.9504	0.5126	1.0000	0.9226	0.9585	0.5346	1.0000
16	20	0.99936	0.9994	0.9997	0.9896	1.0000	0.9996	0.9999	0.9926	1.0000
16	30	0.99999	1.0000	1.0000	0.9998	1.0000	1.0000	1.0000	0.9999	1.0000
32	802	0.90026	0.908	0.940	0.166	1.000	0.026	0.265	0.000	1.000

(a) $\sigma = 0.3$

			normal				*Weibull*			
k	$2q$	$F_{\Omega_i^{2q}}(4)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}	P_i	$P_{0\|i}$	$P_{A\|i}$	P_{Av}
16	4	0.94690	0.9473	0.9388	0.5957	0.9877	0.9826	0.9807	0.8422	0.9990
16	10	0.99982	0.9998	0.9996	0.9939	1.0000	1.0000	1.0000	0.9999	1.0000
16	14	1.00000	1.0000	1.0000	0.9998	1.0000	1.0000	1.0000	1.0000	1.0000
32	8	0.91119	0.9119	0.8572	0.2914	0.9866	0.6847	0.7475	0.0869	0.9976
32	20	0.99925	0.9993	0.9975	0.9544	1.0000	0.9796	0.9756	0.6728	1.0000
32	30	0.99999	1.0000	1.0000	0.9988	1.0000	0.9983	0.9976	0.9422	1.0000
64	28	0.90507	0.908	0.797	0.123	1.000	0.003	0.055	0.000	0.614
64	70	0.99907	0.999	0.996	0.897	1.000	0.015	0.106	0.000	1.000
64	110	0.99999	1.000	1.000	0.998	1.000	0.035	0.142	0.000	1.000

(b) $\sigma = 1.0$

Table 5.9: Probability of convergence when $\varepsilon = 2$ and $i = k/2$, assuming normally distributed delays with $\underset{i \to i+1}{d} = 2.11$, $\mu = 0.34$, $\varrho = 10^{-6}$, and $\lambda = 10$.

	$\sigma = 0.3$		$\sigma = 1.0$	
Distance	$\varepsilon = 1.0$	$\varepsilon = 2.0$	$\varepsilon = 1.0$	$\varepsilon = 2.0$
1	2	1	7	3
2	6	1	12	5
3	18	3	17	8
4	53	5	23	11
5	159	9	29	14
6	503	19	36	18

Table 5.10: Distance away vs. number of inquiries needed, assuming normally distributed delays with $\underset{i \to i+1}{d} = 2.11$, and $\mu = 0.34$.

to 10^{-6} and λ is reduced to 10.

A quick look at Tables 5.8 and 5.9 is enough to determine that the reduction in ϱ and λ has had a considerable effect on the probability of convergence. The correspondence between Tables 5.2 and 5.9b and Tables 5.4 and 5.10b is nearly perfect, showing only a slight reduction in some of the probabilities of convergence. In Table 5.10a it is now possible to have a probability of convergence of 0.99999 when $k = 32$, in Table 5.8a the highest probability of convergence is 0.26556. In Table 5.9a, it is probably possible to get a probability of convergence greater than 0.9 when $k = 32$ (since the maximum q is now 50000), except the computational costs of finding it are too great.

5.2.6 Comparisons

The examples of Sections 5.2.4 and 5.2.5 only compare the algorithm to itself. Any algorithm can look good in comparison with itself, what counts is how well the algorithm looks in comparison with other algorithms which perform the same task. Here the other algorithm of interest is the similar algorithm in [11], assuming single-node inquiry.

The estimation algorithm in [11] simply repeats single-node inquiries until the difference between the time taken for an inquiry and the minimum time required for an inquiry is less than 2ε. If, as above, independent identically distributed hops are assumed, and the node which is the object of the inquiry is h hops away, the probability that any given inquiry will complete quickly enough is $F_{\underset{0 \to 2h}{\chi}}(2\varepsilon)$. If inquiries are independent, the probability that an inquiry with the desired property will be found in p inquiries or less is $1 - (1 - F_{\underset{0 \to 2h}{\chi}}(2\varepsilon))^p$. Table 5.10 shows the number of inquiries needed for nodes from 1 to 6 hops away in order

that the probability of a near-minimal time inquiry being encountered is at least 0.99999.

In order for the estimation algorithm of [11] to be used with peer synchronization, every node must make inquiries of every other node. The total number of inquiries each node must make depends on the distances to the other nodes in the system, which depends on the system architecture. For example, in a 16-node hypercube, for each node there will be 4 nodes 1 hop away, 6 nodes two hops away, 4 nodes 3 hops away, and 1 node 4 hops away. Using Table 5.10, and assuming $\sigma = 0.3$ and $\varepsilon = 1.0$, each node will make 4(2)+6(6)+4(18)+1(53) = 169 inquiries, or a total of $16 \times 169 = 2704$ inquiries system-wide. Since each single-node inquiry requires 2 synchronization messages, containing 3 timestamps, a total of 5408 synchronization messages will be sent, containing 8112 timestamps. Table 5.11 continues these computations, and compares them with the number of synchronization messages and timestamps sent by a k-node inquiry. Equations (5.1) and (5.2) were used to determine the number of synchronization messages and timestamps sent in each k-node inquiry.

The comparisons are made assuming the values of ϱ and λ used in Section 5.2.5. The estimation algorithm of [11] is assumed to be essentially free from the effects of ϱ and λ, though this may not be true in practice. The inquiries N_i sends to N_j must be sent serially. If a large number of inquiries must normally be sent, but a successful inquiry appears early, the resulting estimate must wait until estimates of the remaining nodes are available. If other inquires are not so lucky, the estimate may be forced to wait a long time, long enough that clock drift may have rendered it inaccurate.

The general pattern of the comparison is that the k-node estimation algorithm sends fewer inquiries and synchronization messages (and more timestamps) when σ is large, and sends more of everything when σ is small. The probability of a near-minimal delay path decreases rapidly with path length, eventually there is little chance that the k-node inquiry will complete in near-minimal time, and thus little chance that the interval algorithm will be able to make accurate estimates. Single-node inquiry uses shorter paths, and can thus be used for larger systems, though at considerable cost.

The interval estimation algorithm is therefore usually advantageous when near-minimal delay paths are common, even for long paths. But there are advantages to the interval algorithm besides a reduction in the number of synchronization messages. Perhaps the most important is the structure it brings to the clock distribution process. Because of the k-node inquiry each node sends and receives synchronization messages at regular intervals, synchronization message sends and receives can be scheduled and anticipated. The result

			single-node inquiry			k-node inquiry		
σ	ε	n	# inquiries	# mess	# tstamp	# inquiries	# mess	# tstamp
0.3	1.0	16	2704	5408	8112	∞	∞	∞
0.3	1.0	32	21568	43136	64704	∞	∞	∞
0.3	1.0	64	173696	347392	521088	∞	∞	∞
0.3	2.0	16	432	864	1296	32	992	12032
0.3	2.0	32	2528	5056	7584	∞	∞	∞
0.3	2.0	64	14656	29312	43968	∞	∞	∞
1.0	1.0	16	3056	6112	9168	22	682	8272
1.0	1.0	32	15008	30016	45024	46	2898	69920
1.0	1.0	64	71488	142976	214464	166	21082	1014592
1.0	2.0	16	1360	2720	4080	14	434	5264
1.0	2.0	32	6848	13696	20544	30	1890	45600
1.0	2.0	64	33280	66560	99840	112	14224	684544

(a) $\varrho = 10^{-5}$ and $\lambda = 100$

			single-node inquiry			k-node inquiry		
σ	ε	n	# inquiries	# mess	# tstamp	# inquiries	# mess	# tstamp
0.3	1.0	16	2704	5408	8112	298	9238	112048
0.3	1.0	32	21568	43136	64704	-	-	-
0.3	1.0	64	173696	347392	521088	-	-	-
0.3	2.0	16	432	864	1296	30	930	11280
0.3	2.0	32	2528	5056	7584	-	-	-
0.3	2.0	64	14656	29312	43968	-	-	-
1.0	1.0	16	3056	6112	9168	22	682	8272
1.0	1.0	32	15008	30016	45024	46	2898	69920
1.0	1.0	64	71488	142976	214464	160	20320	977920
1.0	2.0	16	1360	2720	4080	14	434	5264
1.0	2.0	32	6848	13696	20544	30	1890	45600
1.0	2.0	64	33280	66560	99840	110	13970	672320

(b) $\varrho = 10^{-6}$ and $\lambda = 10$

Table 5.11: Comparison of number of inquiries, synchronization messages, and timestamps needed in a hypercube when using single-node inquiry vs. when using k-node inquiry.

is less interference with regular network traffic. In contrast, having each node send its own single-node inquiries causes a mass of synchronization messages, all sent at once, traveling in all directions. Each node may be conducting up to $k-1$ inquiries at once, each of which should complete as quickly as possible, all of them interfering with one another. Trying to limit the number of messages sent at any time will reduce the interference, but increases the likelihood of clock drift becoming a problem. And the examples in Table 5.11 assume a hypercube architecture, which is nearly a best-case situation for single-node inquiry. Other architectures, such as a mesh or ring, would increase both the maximum and average distance between nodes, and greatly increase the number of synchronization messages.

5.3 An Averaging Approach

The method described in Section 5.2 concentrated on the worst case behavior of messages, and made sure that its guesses had a guaranteed uncertainty. This pessimistic approach has the advantage that any guess is guaranteed to be off by no more than its uncertainty, but there is only a *probability* that after $2q$ inquiries that the uncertainty of a particular guess will be less than some given ε. If one were particularly unlucky, one might send $2q$ inquiries and find *none* of the resulting guesses had the desired uncertainty. While increasing q can reduce the likelihood of this happening, the possibility can never be eliminated.

What would happen if, instead of considering worst-case behavior, one used expected case behavior? If the expected time between the creation of T_i and T_0^a is $\mu_{i\to 0}$, then N_0 can compute the "expected" value of α_{i0} as $T_i + \mu_{i\to 0} - T_0^a$. If the value of $\mu_{i\to 0}$ is not known a-priori, it can be estimated from the difference between T_0 and T_0^a.

Each guess is therefore one of these "expected" skews. The system may act unexpectedly during any particular inquiry, so any particular guess may be off by an arbitrary amount. On average, however, the system will act as expected, so guesses can be combined by averaging them, and the result should be close to the true skew. As the number of guesses increases, the probability that their average is within ε of the true skew also increases. The average becomes an estimate when enough inquiries have been sent so that the probability that the average is within ε of the true skew is greater than some specified value, called the *probability of validity*, usually 0.999 or 0.99999 or some other value close to 1. The number of inquiries needed is a constant that depends primarily on the variance of synchronization

message delays. In contrast to the interval algorithm, the uncertainty of an estimate is not guaranteed, but the number of inquiries needed to make an estimate is.

This algorithm is similar to the one described by Arvind [2]. However, [2] focuses on a master/slave system where synchronization messages are sent from master to slave only. This does not allow $\underset{i\to j}{\mu}$ to be deduced from round-trip times, and thus is vulnerable to fluctuations in message delay characteristics. Some mention is made of using single-node inquiry, and deducing $\underset{i\to j}{\mu}$ from the total time taken, but this is not fully explored. Little is also said about the effects of clock drift during the times the messages are being sent.

5.3.1 An Average Guess

While the estimation algorithm may be different, distribution of clock information is identical, therefore the same notation used for the interval algorithm will be used here. As for the interval algorithm, the estimation process will be considered from the point of view of N_0, since any node can become N_0 by simply re-labeling the nodes. And it is still assumed that $2q$ inquiries are done, alternating direction each time, and that the start times of consecutive inquiries are λ apart.

Calculation of the individual guesses is straightforward. If inquiry h is a forward inquiry, then the guess for the skew of N_i with respect to N_0, $\tilde{\alpha}^h_{i|0}$, is:

$$\tilde{\alpha}^h_{i|0} = T^h_i + \underset{i\to k}{\mu^h} + \underset{i\to k}{W^h} - T^h_0 \tag{5.36}$$

The corresponding equation for backward inquiries is:

$$\tilde{\alpha}^h_{i|0} = T^h_i + \underset{i\to 0}{\mu^h} + \underset{i\to 1}{W^h} - T^h_0 \tag{5.37}$$

The estimate derived from the $2q$ inquiries is the average of the guesses:

$$\tilde{\alpha}_{i|0} = \frac{1}{2q}\sum_{p=0}^{q-1}\left(T^{2p}_i + T^{2p+1}_i + \underset{i\to k}{\mu^{2p}} + \underset{i\to 0}{\mu^{2p+1}} + \underset{i\to k}{W^{2p}} + \underset{i\to 0}{W^{2p+1}} - T^{2p}_k - T^{2p+1}_0\right) \tag{5.38}$$

In some cases there may be no mean delay time. At any time there is a mean delay, but its value is not known and cannot be predicted. Burst loads or highly variable loads are the usual reason for such behavior. In such cases the mean delay may be computed (or estimated) from the round-trip time. For each i, $0 < i < k$, two constants are specified:

$\underset{i\to k}{\zeta}$: The expected ratio of $\left(\underset{i\to k}{d} + \underset{i\to k}{\mathcal{X}}\right) / \left(\underset{0\to k}{d} + \underset{0\to k}{\mathcal{X}}\right)$.

$\underset{i\to 0}{\zeta}$: The expected ratio of $\left(\underset{i\to 0}{d} + \underset{i\to 0}{\mathcal{X}}\right) / \left(\underset{k\to 0}{d} + \underset{k\to 0}{\mathcal{X}}\right)$.

Then $\underset{i\to k}{\mu}^h \approx \underset{i\to k}{\zeta}\,(T_k^h - T_0^h)$ and $\underset{i\to 0}{\mu}^h \approx \underset{i\to 0}{\zeta}\,(T_0^h - T_k^h)$. The final estimate is then:

$$\begin{aligned}\tilde{\alpha}_{i\hat{l}0} &= \frac{1}{2q}\sum_{p=0}^{q-1}\left(T_i^{2p} + T_i^{2p+1}\right) + \frac{\underset{i\to k}{\zeta}}{2q}\sum_{p=0}^{q-1}\left(T_k^{2p} - T_0^{2p} - \underset{0\to k}{W}^{2p}\right) \\ &\quad + \frac{\underset{i\to 0}{\zeta}}{2q}\sum_{p=0}^{q-1}\left(T_0^{2p+1} - T_k^{2p+1} - \underset{k\to 0}{W}^{2p+1}\right) - \frac{1}{2q}\sum_{p=0}^{q-1}\left(\underset{i\to k}{W}^{2p} + \underset{i\to 0}{W}^{2p+1}\right) \\ &\quad - \frac{1}{2q}\sum_{p=0}^{q-1}\left(T_k^{2p} + T_0^{2p+1}\right) \end{aligned} \tag{5.39}$$

Both the individual guesses and the final estimate are probably close to the actual skew value, but there are no guarantees. There are two main sources of inaccuracy in both the guesses and the estimate. *Inherent error* is the result of clock drift both during and between inquiries. Its value cannot be determined a-priori, but must be computed along with the guess or estimate. However, a probable upper bound may be computed ahead of time. In general, the inherent error of the estimate is greater than that of the guesses, and increases as q increases. *Inherent uncertainty* is the result of the variability of message delays, and the resulting difference between $\underset{i\to k}{\mu}^h$ and $\underset{i\to 0}{\mu}^h$ and the actual delay. Its value can be computed a-priori, and is such that, assuming the inherent error is zero, the difference between the actual skew value and the guess or estimate is less than the inherent uncertainty, with probability greater than or equal to the probability of validity. The inherent uncertainty of the estimate is usually lower than that of any guess, and it does decrease as q increases. The uncertainty of a guess or estimate is the sum of its inherent error and inherent uncertainty, and the remainder of this sections shows how to find a q such that the uncertainty of the estimate is less than the specified maximum uncertainty, ε.

5.3.2 Inherent Error

There are two sources of inherent error in $\tilde{\alpha}_{i\hat{l}0}^h$: clock drift during and after inquiry h, and the computation of $\underset{i\to k}{\mu}^h$ or $\underset{i\to 0}{\mu}^h$. The inherent error of the estimate is the average of the inherent errors of the guesses.

Error from Clock Drift

Ideally, for forward inquiries $\tilde{\alpha}_{i\hat{l}0}^h = T_i^h + \underset{i\to k}{d} + \underset{i\to k}{\mathcal{X}}^h + \underset{i\to k}{\mathcal{W}}^h - T_k^h$. However, since $\underset{i\to k}{d} + \underset{i\to k}{\mathcal{X}}^h$ is not known, $\underset{i\to k}{\mu}^h$ is used instead, and since $\underset{i\to k}{\mathcal{W}}^h$ is not known, $\underset{i\to k}{W}^h$ is used instead. To get

an idea of how much error is introduced by the use of $W^h_{i\to k}$, consider the following:

$$\begin{array}{rcccl}
\mathcal{W}^h_{i\to k}(1-\varrho) & \leq & W^h_{i\to k} & \leq & \mathcal{W}^h_{i\to k}(1+\varrho) \\
\frac{W^h_{i\to k}}{1+\varrho} & \leq & \mathcal{W}^h_{i\to k} & \leq & \frac{W_{h\ i\to k}}{1-\varrho} \\
\frac{W^h_{i\to k}(1+\varrho)}{1-\varrho^2} & \leq & \mathcal{W}^h_{i\to k} & \leq & \frac{W^h_{i\to k}(1-\varrho)}{1-\varrho^2} \\
W^h_{i\to k}-\varrho W^h_{i\to k} & \leq & \mathcal{W}^h_{i\to k} & \leq & W^h_{i\to k}+\varrho W^h_{i\to k}
\end{array}$$

Where the final step involves dropping the ϱ^2 term in the denominators. Thus, the inherent error introduced by using $W^h_{i\to k}$ is $\varrho W^h_{i\to k}$. This still does not account for clock drift after the inquiry has completed, and before the estimate is computed. This adds another $2\varrho\lambda(2q-h-1)$ to the inherent error of $\tilde{\alpha}^h_{i|0}$.

Error from Computation of $\mu^h_{i\to k}$ and $\mu^h_{i\to 0}$

When estimated means are to be used, one must specify the variances of $\mathcal{X}_{0\to k}$ and $\mathcal{X}_{k\to 0}$, $\sigma^2_{0\to k}$ and $\sigma^2_{k\to 0}$, chosen large enough to allow for any likely variation during system operation.

Consider the case of forward inquiries first. Consider the relationship between actual and measured times, in this case $\mathcal{W}^h_{0\to k}(1-\varrho) \leq W^h_{0\to k} \leq \mathcal{W}^h_{0\to k}(1+\varrho)$. Therefore, if ϱ^2 terms are ignored, then $W^h_{0\to k}(1-2\varrho) \leq \mathcal{W}^h_{0\to k}(1-\varrho)$ and $\mathcal{W}^h_{0\to k}(1+\varrho) \leq W^h_{0\to k}(1+2\varrho)$. It then follows that:

$$\begin{array}{rcccl}
\left(d_{0\to k}+\mathcal{X}^h_{0\to k}+\mathcal{W}^h_{0\to k}\right)(1-\varrho) & \leq & T^h_k-T^h_0 & \leq & \left(d_{0\to k}+\mathcal{X}^h_{0\to k}+\mathcal{W}^h_{0\to k}\right)(1+\varrho) \\
\left(d_{0\to k}+\mathcal{X}^h_{0\to k}\right)(1-\varrho) + W^h_{0\to k}(1-2\varrho) & \leq & T^h_k-T^h_0 & \leq & \left(d_{0\to k}+\mathcal{X}^h_{0\to k}\right)(1+\varrho) + W^h_{0\to k}(1+2\varrho) \\
\left(d_{0\to k}+\mathcal{X}^h_{0\to k}\right)(1-\varrho)-2\varrho W^h_{0\to k} & \leq & T^h_k-T^h_0-W^h_{0\to k} & \leq & \left(d_{0\to k}+\mathcal{X}^h_{0\to k}\right)(1+\varrho)+2\varrho W^h_{0\to k}
\end{array}$$

Since $\mu^h_{i\to k} = \zeta_{i\to k}\left(T^h_k-T^h_0-W^h_{0\to k}\right)$, the inherent error added to the guess from the computation of $\mu^h_{i\to k}$ is $\pm\zeta_{i\to k}\varrho\left(d_{0\to k}+\mathcal{X}^h_{0\to k}+2W^h_{0\to k}\right)$. For backward inquiries the inherent uncertainty will be $\pm\zeta_{i\to k}\varrho\left(d_{k\to 0}+\mathcal{X}^h_{k\to 0}+2W^h_{k\to 0}\right)$.

Inherent Error of the Final Estimate

Given all the above calculations, a formula for the inherent error of the final estimate, e, can be derived. Equation (5.40) is the formula to use when $\mu^h_{i\to k}$ and $\mu^h_{i\to 0}$ are constants. Equation (5.41) is the formula to use when $\mu^h_{i\to k}$ and $\mu^h_{i\to 0}$ are calculated.

$$e = \varrho\lambda(2q-1) + \frac{\varrho}{2q}\sum_{p=0}^{q-1}\left(W_{i\to k}^{2p} + W_{i\to 0}^{2p+1}\right) \tag{5.40}$$

$$\begin{aligned} e = {} & \varrho\lambda(2q-1) + \frac{\varrho}{2q}\sum_{p=0}^{q-1}\left(W_{i\to k}^{2p} + W_{i\to 0}^{2p+1}\right) + \frac{\varrho_{i\to k}\zeta}{2q}\sum_{p=0}^{q-1}\left(d_{0\to k} + \mathcal{X}_{0\to k}^{2p} + 2W_{0\to k}^{2p}\right) \\ & + \frac{\varrho_{i\to 0}\zeta}{2q}\sum_{p=0}^{q-1}\left(d_{k\to 0} + \mathcal{X}_{k\to 0}^{2p+1} + 2W_{k\to 0}^{2p}\right) \end{aligned} \tag{5.41}$$

In both equations the inherent error depends on a number of variables whose values are not known beforehand. This makes it impossible to predict the inherent error of the estimate. However, an upper bound on the inherent error would serve just as well, and one can be found by replacing $\mathcal{X}_{i\to k}^{h}$, $\mathcal{X}_{i\to 0}^{h}$, $W_{i\to k}^{h}$, and $W_{i\to 0}^{h}$ in Equations (5.40) and (5.41) with their respective upper bounds. In general, no absolute upper bound may exist for these values, but highly probable upper bounds can usually be determined, yielding a highly probable upper bound for the inherent error. By choosing upper bounds that are probable enough, one can compute an upper bound for the inherent error for any specified probability.

5.3.3 Inherent Uncertainty

The inherent uncertainty of a guess or estimate is entirely due to the variability of message delays, and is essentially a bound on the error so introduced. Because there is usually no absolute upper bound on message delays, there can be no absolute upper bound on the error they introduce. For this reason the inherent uncertainty always has an associated probability, such that the probability the error caused by uncertainty in message delays is no greater than the inherent uncertainty, is greater than or equal to the specified probability.

The inherent uncertainty when constant means are used can be found from an examination of Equation (5.38). The right side of the equation can be re-organized as follows:

$$\frac{1}{2q}\sum_{p=0}^{q-1}\left[\left(T_i^{2p} - T_k^{2p} - W_{i\to k}^{2p} + \mu_{i\to k}\right) + \left(T_i^{2p+1} - T_0^{2p+1} - W_{i\to 0}^{2p+1} + \mu_{i\to 0}\right)\right]$$

The estimate itself is then the sum of $2q$ terms. For each term, one of the following approximations will hold:

$$\begin{aligned} T_i^h - T_k^h - W_{i\to k}^h &\approx \alpha_{il0} + d_{i\to k} + \mathcal{X}_{i\to k}^h \\ T_i^h - T_0^h - W_{i\to 0}^h &\approx \alpha_{il0} + d_{i\to 0} + \mathcal{X}_{i\to 0}^h \end{aligned}$$

Where the approximation is a result of using $\underset{i\to k}{W^h}$ or $\underset{i\to 0}{W^h}$ instead of $\underset{i\to k}{\mathcal{W}^h}$ or $\underset{i\to 0}{\mathcal{W}^h}$, and any error introduced by this approximation has been accounted for by the inherent error. So, if $\underset{i\to k}{\mu}$ and $\underset{i\to 0}{\mu}$ are equal to the actual means of $\underset{i\to k}{d} + \underset{i\to k}{\mathcal{X}}$ and $\underset{i\to 0}{d} + \underset{i\to 0}{\mathcal{X}}$, then the mean value of each term will be $\alpha_{i|0}$. The variance of each term is the variance of either $\underset{i\to k}{\mathcal{X}}$ or $\underset{i\to 0}{\mathcal{X}}$, which should be equal to either $\underset{i\to k}{\sigma^2}$ or $\underset{i\to 0}{\sigma^2}$.

A similar re-arrangement can be done for the right-hand side of Equation (5.39):

$$\frac{1}{2q}\sum_{p=0}^{q-1} \left[\left(T_i^{2p} - T_k^{2p} - \underset{i\to k}{W^{2p}} + \underset{i\to k}{\zeta}\left(T_k^{2p} - T_0^{2p} - \underset{0\to k}{W^{2p}}\right)\right) + \left(T_i^{2p+1} - T_0^{2p+1} - \underset{i\to 0}{W^{2p+1}} + \underset{i\to 0}{\zeta}\left(T_0^{2p+1} - T_k^{2p+1} - \underset{k\to 0}{W^{2p+1}}\right)\right)\right]$$

Each term will have a mean of $\alpha_{i|0}$ if $\underset{i\to k}{\zeta}$ and $\underset{i\to 0}{\zeta}$ are equal to the actual means of $\left(\underset{i\to k}{d} + \underset{i\to k}{\mathcal{X}}\right) / \left(\underset{0\to k}{d} + \underset{0\to k}{\mathcal{X}}\right)$ and $\left(\underset{i\to 0}{d} + \underset{i\to 0}{\mathcal{X}}\right) / \left(\underset{k\to 0}{d} + \underset{k\to 0}{\mathcal{X}}\right)$. The variance of each term will be the variance of either the $\underset{i\to k}{\zeta}$ or $\underset{i\to 0}{\zeta}$ term, which will be either $\underset{i\to k}{\zeta}\,\underset{0\to k}{\sigma^2}$ or $\underset{i\to 0}{\zeta}\,\underset{k\to 0}{\sigma^2}$.

Since an estimate is the sum of a number of independent random variables, application of the central limit theorem is tempting. One might hesitate due to the bad experience with this theorem in Section 5.2.4, but there is good reason it may work better here. In Section 5.2.4 the number of variables summed was either $k - i$ or i, which was apparently too small for the approximation to work, especially for large variances. In this case $2q$ variables are being summed, and each of them is the sum of $k - i$ or i variables, for a total of kq independent random variables. Moreover, it should be intuitively obvious that as the variance of delays increases, the variability of the guesses will increase, so q will increase. The greater number of variables (increasing with the variance) in the sum means that the approximation of the central limit theorem is more likely to be valid.

Application of the central limit theorem makes it very easy to predict the mean and variance of an estimate. The mean of the sum will be the sum of the means, and the variance of the sum will be the sum of the variances. The value of $\tilde{\alpha}_{i|0}$ will be approximately normally distributed with mean $\alpha_{i|0}$ and variance $\left(\underset{i\to k}{\sigma^2} + \underset{i\to 0}{\sigma^2}\right)/4q$ if constant means are used, or variance $(\underset{i\to k}{\zeta}\,\underset{0\to k}{\sigma^2} + \underset{k\to 0}{\zeta}\,\underset{k\to 0}{\sigma^2})/4q$ if calculated means are used. If $P_\epsilon(x)$ is the probability that the error introduced by uncertainty in message delays is less than x, then for constant means:

$$P_\epsilon(x) = Erf\left(\frac{x\sqrt{2q}}{\sqrt{\underset{i\to k}{\sigma^2} + \underset{i\to 0}{\sigma^2}}}\right)$$

For calculated means:

$$P_\epsilon(x) = Erf\left(\frac{x\sqrt{2q}}{\sqrt{\underset{i\to k}{\zeta}\ \underset{0\to k}{\sigma^2} + \underset{i\to 0}{\zeta}\ \underset{k\to 0}{\sigma^2}}}\right)$$

If Erf^{-1} is the inverse error function, then for a given x, a function mapping probability to the value of q can be found.

$$q = \frac{\underset{i\to k}{\sigma^2} + \underset{i\to 0}{\sigma^2}}{2x^2}\left(Erf^{-1}\left(P_\epsilon(x)\right)\right)^2 \tag{5.42}$$

$$q = \frac{\underset{i\to k}{\zeta}\ \underset{0\to k}{\sigma^2} + \underset{i\to 0}{\zeta}\ \underset{k\to 0}{\sigma^2}}{2x^2}\left(Erf^{-1}\left(P_\epsilon(x)\right)\right)^2 \tag{5.43}$$

Equation (5.42) is used for constant mean, and Equation (5.43) is used for estimated means.

The inherent uncertainty of an estimate is the value of x such that $P_\epsilon(x)$ is equal to the probability of validity. To find minimum q so that the *uncertainty* of an estimate is less than ε, set x to be ε minus the inherent error[3], set $P_\epsilon(x)$ to be the probability of validity, and use either Equation (5.42) or Equation (5.43). If one wishes to allow for the probability associated with the inherent error, set $P_\epsilon(x)$ to be the probability of invalidity divided by the probability for the inherent error.

5.3.4 Examples with $\varrho = 0$

As for the interval algorithm, the first set of examples assume $\varrho \approx 0$. This assumption simplifies calculations since the inherent error is 0. Assuming individual delays are normally distributed causes no problems for the averaging algorithm, in large part because it makes no assumptions about absolute minimum delay times[4]. Even so, simulations were run using the Weibull distribution. Both distributions assumed $\mu = 2.11 + 0.34 = 2.45$, and the same standard deviations as before, $\sigma = 1.0$ or $exsig = 0.3$. Tables 5.12 and 5.13 summarize the results.

Since all delays are assumed independent and identically distributed, $\underset{i\to k}{\sigma^2} = (k-i)\sigma^2$, $\underset{i\to 0}{\sigma^2} = i\sigma^2$, $\underset{0\to k}{\sigma^2} = \underset{k\to 0}{\sigma^2} = k\sigma^2$, $\underset{i\to k}{\zeta} = (k-i)/k$, and $\underset{i\to 0}{\zeta} = i/k$. Equations (5.42) and (5.43) therefore reduce to the following:

[3]Computation of x is complicated by the fact that the inherent error depends on q. A simple iterative procedure, described in Section 5.3.5, can be used to resolve this.

[4]Realistically, the absolute minimum delay is 0. However, the averaging algorithm is unaffected by negative delays, and theoretically would work even if all delays were negative.

			normal			*Weibull*		
k	$2q$	$P_\epsilon(1)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_i	$P_{0\|i}$	$P_{A\|i}$
16	2	0.90442	0.9038	0.7153	0.1440	0.9073	0.7373	0.1779
16	8	0.99914	0.9989	0.9943	0.9552	0.9985	0.9930	0.9514
16	16	0.99999	1.0000	1.0000	0.9996	1.0000	0.9999	0.9990
32	4	0.90442	0.9042	0.6675	0.0793	0.9053	0.6696	0.0856
32	16	0.99914	0.9990	0.9934	0.9418	0.9990	0.9930	0.9396
32	30	0.99999	1.0000	1.0000	0.9992	0.9997	0.9985	0.9942
64	8	0.90442	0.9044	0.6286	0.0474	0.9025	0.6254	0.0466
64	32	0.99914	0.9992	0.9923	0.9242	0.9991	0.9919	0.9222
64	58	0.99999	1.0000	0.9999	0.9985	1.0000	0.9999	0.9984

(a) $\sigma = 0.3$

			normal			*Weibull*		
k	$2q$	$P_\epsilon(1)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_i	$P_{0\|i}$	$P_{A\|i}$
16	22	0.90275	0.9034	0.7153	0.1433	0.9173	0.7876	0.2742
16	88	0.99909	0.9991	0.9947	0.9580	0.9975	0.9907	0.9433
16	158	0.99999	1.0000	0.9999	0.9992	0.9999	0.9994	0.9957
32	44	0.90275	0.9030	0.6601	0.0757	0.9064	0.6905	0.1104
32	174	0.99903	0.9990	0.9929	0.9363	0.9986	0.9915	0.9324
32	314	0.99999	1.0000	0.9999	0.9987	1.0000	0.9998	0.9974
64	88	0.90275	0.9028	0.6226	0.0452	0.9035	0.6379	0.0556
64	348	0.99903	0.999	0.992	0.918	0.999	0.991	0.919
64	626	0.99999	1.000	1.000	0.998	1.000	1.000	0.998

(b) $\sigma = 1.0$

Table 5.12: Probability of validity when $\varepsilon = 1.0$, with $\mu = 2.45$.

			normal			*Weibull*		
k	$2q$	$P_\epsilon(2)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_i	$P_{0\|i}$	$P_{A\|i}$
16	2	0.99914	0.9991	0.9951	0.9610	0.9977	0.9908	0.9429
16	4	0.99999	1.0000	1.0000	0.9996	0.9999	0.9997	0.9973
32	2	0.98158	0.9799	0.9102	0.5172	0.9805	0.9156	0.5598
32	4	0.99914	0.9989	0.9933	0.9413	0.9984	0.9918	0.9368
32	8	0.99999	1.0000	1.0000	0.9996	1.0000	0.9999	0.9988
64	2	0.90442	0.9038	0.6288	0.0478	0.9037	0.6351	0.0544
64	8	0.99914	0.9991	0.9925	0.9254	0.9990	0.9919	0.9236
64	16	0.99999	1.0000	1.0000	0.9994	1.0000	1.0000	0.9992

(a) $\sigma = 0.3$

			normal			*Weibull*		
k	$2q$	$P_\epsilon(2)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_i	$P_{0\|i}$	$P_{A\|i}$
16	6	0.91674	0.9169	0.7510	0.1914	0.9406	0.8902	0.5943
16	22	0.99909	0.9990	0.9943	0.9562	0.9946	0.9855	0.9254
16	40	0.99999	1.0000	0.9999	0.9992	0.9995	0.9983	0.9891
32	12	0.91674	0.9168	0.7003	0.1062	0.9280	0.7803	0.2419
32	44	0.99909	0.9990	0.9933	0.9393	0.9974	0.9884	0.9262
32	80	0.99999	1.0000	0.9999	0.9989	0.9936	0.9848	0.9566
64	22	0.90275	0.9033	0.6245	0.0450	0.9068	0.6635	0.0783
64	88	0.99909	0.9991	0.9921	0.9223	0.9987	0.9904	0.9204
64	158	0.99999	1.000	1.000	0.998	1.000	1.000	0.996

(b) $\sigma = 1.0$

Table 5.13: Probability of validity when $\varepsilon = 2.0$, with $\mu = 2.45$.

$$q = \frac{k\sigma^2}{2x^2}\left(Erf^{-1}\left(P_\epsilon(x)\right)\right)^2 \tag{5.44}$$

Which does not depend on i. Therefore, the tables have no column for i, and results are not presented for particular nodes. Also, since all estimates generated by the averaging algorithm have the same uncertainty, there is no notion of the average uncertainty being less than ε. Thus there is no column for P_{Av}. The remaining columns are the same as for the interval algorithm.

The analytical and simulation results for normal distribution match up very closely. However, while the simulation results for the normal and Weibull distributions are quite close, there is a general pattern to their differences. A careful comparison shows that when q is small, and P_i is about 0.99 or less, P_i will be slightly greater for the Weibull distribution than for the normal distribution. The situation reverses when q becomes large enough that P_i is greater than 0.99, the normal distribution yields a higher P_i than the Weibull distribution. The same observations may be made for $P_{0|i}$ and $P_{A|i}$. And in all cases the effect is more pronounced when $\sigma = 1.0$.

The close match between simulation and analytical results confirms that the central limit theorem does provide a good approximation in this case. The close match between normal and Weibull distribution shows that the shape of the distribution does not matter, only its variance.

Finally, the simulation results verify the relationships between q, k, and σ seen in Equation (5.44). The relationship between q and k should be linear, and as k doubles from 16 to 32, and then 32 to 64, the values of q also double. The relationship between q and σ^2 should also be linear. The ratio $1.0^2/0.3^2$ is approximately 11, so q should increase by a factor of 11 as σ changes from 0.3 to 1.0. This is also the case, there are a few exceptions, which are apparently due to the granularity of q (which must be an integer), and the number of inquiries (which is twice q).

5.3.5 Examples with $\varrho > 0$

Assuming $\varrho = 0$ causes inherent error to be neglected. Normally, ϱ will be very small, and since all the inherent error terms in Equations (5.40) and (5.41) contain ϱ, the inherent error should be small as well. This is similar to the argument used for the interval algorithm, which turned out to be incorrect. Therefore, this section provides examples in which the effects of inherent error are taken into account.

The same values of ϱ and λ are used as in Section 5.2.5, so for the first set of examples $\varrho = 10^{-5}$ and $\lambda = 100$. The values of μ and σ are the same as above. Estimated means are assumed to be used since the inherent error of Equation (5.41) is greater than that of Equation (5.40), and thus will provide more of a worst-case example. The inherent error of Equation (5.41) depends on the value of $\underset{0\to k}{d} + \underset{0\to 0}{x}$, so a maximum for this value must be supplied, for these examples the maximum is assumed to be 6msec.

Examination of Equation (5.41) shows that if individual delays are independent and identically distributed, then inherent error (like inherent uncertainty) does not depend on i. Equation (5.44) may still be used to determine q, only $\varepsilon - e$ is used in place of ε. A minor difficulty arises because e depends on q. This can be resolved through simple iteration. Assume $e = 0$ to start, and compute q. Find e for the computed q, and re-compute q for the new value of e. If the two values of q are not equal, re-compute e for the new q, and re-compute q for the new e. Continue until successive q's are equal, or until $e \geq \varepsilon$. Since each e is greater than the previous one, each q is at least as large as the previous one, so one of the termination conditions is guaranteed to occur.

The results are summarized in Tables 5.14 and 5.15. Another column has been added, e, which shows the inherent error for the specified value of q. Note also that $P_\epsilon(\varepsilon - e)$ is calculated instead of $P_\epsilon(\varepsilon)$.

The value of inherent error for a given q is dominated by the $\varrho\lambda(2q-1)$ term (at least when the node wait is 0). When $\varepsilon = 1$ the "maximum" q is 500, any larger value of q will cause an inherent error larger than ε. However, the maximum probability of validity occurs when q is much less than 500, when $q = 167$ in fact. This is because while the probability of validity increases with q, the rate of increase slows for large q. The value of e, on the other hand, continues to increase linearly with q, eventually overwhelming the increase in the probability of validity. The probability of validity when $q = 167$ is shown in Table 5.15b in cases where it is below 0.99999.

Accounting for inherent error has eliminated the nice linear relationship between q and k, and q and σ^2. This is especially noticeable for higher probabilities of validity, and larger values of q. In Table 5.15b, one can see that for a probability of validity of 0.9, q increases from 12 to 24 to 55 as k is doubled. When the probability of validity is 0.999, q goes from 55 to ∞ with only one doubling of k. In the case of σ^2, where before an increase in σ from 0.3 to 1.0 would increase q by approximately 11 times, the increase is now much greater, infinitely so in some cases. This is most easily seen in Table 5.14, as the values of q in

				normal			*Weibull*		
k	$2q$	e	$P_\epsilon(1-e)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_i	$P_{0\|i}$	$P_{A\|i}$
16	2	0.00148	0.90393	0.9040	0.7150	0.1425	0.9070	0.7369	0.1771
16	8	0.00748	0.99906	0.9990	0.9948	0.9581	0.9987	0.9935	0.9531
16	16	0.01548	1.00000	1.0000	1.0000	0.9997	1.0000	0.9999	0.9992
32	4	0.00396	0.90310	0.7212	0.4878	0.0389	0.9029	0.6655	0.0814
32	18	0.01796	0.99948	0.9995	0.9960	0.9615	0.9994	0.9957	0.9598
32	30	0.02996	0.99999	1.0000	0.9999	0.9988	1.0000	0.9999	0.9985
64	8	0.00892	0.90142	0.9021	0.6216	0.0443	0.9002	0.6190	0.0439
64	34	0.03492	0.99909	0.999	0.992	0.923	0.9990	0.9916	0.9200
64	66	0.06692	0.99999	1.000	1.000	0.998	1.000	1.000	0.998

(a) $\sigma = 0.3$

				normal			*Weibull*		
k	$2q$	e	$P_\epsilon(1-e)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_i	$P_{0\|i}$	$P_{A\|i}$
16	24	0.02348	0.90924	0.9093	0.7272	0.1568	0.9213	0.7946	0.2863
16	110	0.10948	0.99904	0.9990	0.9946	0.9572	0.9978	0.9911	0.9446
16	334	0.33348	0.99998	1.0000	0.9999	0.9988	1.000	1.000	0.997
16	∞	∞	0.99999	-	-	-	-	-	-
32	48	0.04796	0.90085	0.9009	0.6565	0.0730	0.9052	0.6869	0.1053
32	334	0.33396	0.99766	0.998	0.984	0.875	0.997	0.984	0.878
32	∞	∞	0.99900	-	-	-	-	-	-
32	∞	∞	0.99999	-	-	-	-	-	-
64	110	0.11092	0.90073	0.903	0.620	0.044	0.9016	0.6321	0.0521
64	334	0.33492	0.96834	0.968	0.840	0.296	0.9685	0.8454	0.3085
64	∞	∞	0.99900	-	-	-	-	-	-
64	∞	∞	0.99999	-	-	-	-	-	-

(b) $\sigma = 1.0$

Table 5.14: Probability of validity when $\varepsilon = 1.0$, with $\mu = 2.45$, $\varrho = 10^{-5}$, and $\lambda = 100$.

				normal			*Weibull*		
k	$2q$	e	$P_\epsilon(2-e)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_i	$P_{0\|i}$	$P_{A\|i}$
16	2	0.00148	0.99913	0.9991	0.9951	0.9606	0.9977	0.9908	0.9428
16	4	0.00348	1.00000	1.0000	1.0000	0.9998	1.0000	0.9998	0.9981
32	2	0.00196	0.98146	0.9815	0.9088	0.5155	0.9809	0.9151	0.5557
32	4	0.00396	0.99912	0.9991	0.9935	0.9414	0.9987	0.9921	0.9370
32	8	0.00796	1.00000	1.0000	1.0000	0.9996	1.0000	0.9999	0.9990
64	2	0.00292	0.90399	0.9044	0.6282	0.0474	0.9036	0.6352	0.0536
64	8	0.00892	0.99909	0.9991	0.9920	0.9228	0.9990	0.9917	0.9213
64	16	0.01692	1.00000	1.0000	1.0000	0.9993	1.0000	0.9999	0.9992

(a) $\sigma = 0.3$

				normal			*Weibull*		
k	$2q$	e	$P_\epsilon(2-e)$	P_i	$P_{0\|i}$	$P_{A\|i}$	P_i	$P_{0\|i}$	$P_{A\|i}$
16	6	0.00548	0.91589	0.9158	0.7435	0.1770	0.9403	0.8878	0.5831
16	24	0.02348	0.99938	0.9994	0.9964	0.9703	0.9959	0.9882	0.9374
16	42	0.04148	0.99999	1.0000	0.9999	0.9994	0.9996	0.9985	0.9904
32	12	0.01196	0.91487	0.9150	0.6934	0.1011	0.9262	0.7756	0.2350
32	46	0.04596	0.99908	0.9991	0.9933	0.9390	0.9977	0.9889	0.9271
32	86	0.08596	0.99999	1.0000	0.9999	0.9987	1.000	0.999	0.994
64	24	0.02492	0.91282	0.9132	0.6535	0.0612	0.917	0.689	0.098
64	96	0.09692	0.99902	0.999	0.991	0.917	0.999	0.991	0.917
64	192	0.19292	0.99999	1.000	1.000	0.998	1.000	1.000	0.997

(b) $\sigma = 1.0$

Table 5.15: Probability of validity when $\varepsilon = 2.0$, with $\mu = 2.45$, $\varrho = 10^{-5}$, and $\lambda = 100$.

Table 5.15 are too small.

The averaging algorithm also has difficulty making estimates when ϱ is non-zero. Again the most trouble occurs when q is large, and is the result of clock drift over the period when the inquiries are being sent. One possible solution is to reduce q, which means either reducing k, or increasing ε. Usually, k is fixed, and ε may be bounded by the needs of the adjustment algorithm. Another possibility is to reduce either ϱ or λ. This will not reduce q, but will reduce the inherent error by either reducing the clock drift over the period of time when the inquiries are being sent, or by reducing the period of time when the inquiries are being sent. Tables 5.16 and 5.17 show the results if $\varrho = 10^{-6}$ and $\lambda = 10$msec. Since the results for the normal and Weibull distributions do not differ significantly the results for the Weibull distribution are omitted.

An immediate result of reducing ϱ to 10^{-6} and λ to 10 is that probabilities of validity that were previously impossible are now possible. With the larger values of ϱ and λ, when $exsig = 1.0$, and $\varepsilon = 1$, one can not achieve a probability of validity of 0.99999, even when $k = 16$. With the smaller values of ϱ and λ, a probability of validity of 0.99999 can be achieved not only when $k = 16$, but when $k = 32$, and even when $k = 64$.

The lower values of ε and λ have reduced the effects of inherent error to the point where the values in Tables 5.16 and 5.17 are almost identical to those in Tables 5.12 and 5.13. The few differences are when q is large and inherent error has a chance to have an effect. Even in these cases q is increased by no more than 4.

5.3.6 Comparisons

Since the averaging algorithm is similar to the algorithm in [2], it is only logical that the two should be compared. Both have very similar characteristics, and differ primarily in their method of distributing clock information.

The estimation algorithm in [2], like the one in [11], uses single-node inquiry to distribute clock information, with one twist. In [2] it is assumed that the mean delay is known beforehand, and does not change significantly. As a result, "half" inquiries are done, instead of N_i sending a synchronization message to N_j and having N_j send one back, N_j simply sends a synchronization message to N_i, and N_i makes a guess based on the timestamp and the mean delay. While this cuts the number of synchronization messages in half, it is vulnerable to changes in mean delay. And changes in mean delay are quite possible, especially if all nodes are conducting inquiries at the same time.

				normal		
k	$2q$	e	$P_c(1-e)$	P_i	$P_{0\|i}$	$P_{A\|i}$
16	2	0.00006	0.90440	0.9045	0.7152	0.1435
16	8	0.00012	0.99914	0.9991	0.9951	0.9610
16	16	0.00020	1.00000	1.0000	1.0000	0.9998
32	4	0.00013	0.90438	0.9040	0.6644	0.0783
32	16	0.00025	0.99914	0.9992	0.9936	0.9420
32	30	0.00039	0.99999	1.0000	1.0000	0.9992
64	8	0.00026	0.90433	0.9042	0.6282	0.0474
64	32	0.00050	0.99914	0.9991	0.9921	0.9249
64	58	0.00076	0.99999	1.0000	0.9999	0.9985

(a) $\sigma = 0.3$

				normal		
k	$2q$	e	$P_c(1-e)$	P_i	$P_{0\|i}$	$P_{A\|i}$
16	22	0.00026	0.90266	0.9027	0.7114	0.1395
16	88	0.00092	0.99908	0.9991	0.9948	0.9585
16	158	0.00162	0.99999	1.0000	0.9999	0.9993
32	44	0.00053	0.90257	0.9026	0.6599	0.0750
32	174	0.00183	0.99900	0.999	0.993	0.936
32	316	0.00325	0.99999	1.000	1.000	0.999
64	88	0.00106	0.90239	0.902	0.624	0.045
64	350	0.00368	0.99902	0.999	0.991	0.916
64	634	0.00652	0.99999	1.000	1.000	0.998

(b) $\sigma = 1.0$

Table 5.16: Probability of validity when $\varepsilon = 1.0$, with $\mu = 2.45$, $\varrho = 10^{-6}$, and $\lambda = 10$.

				normal		
k	$2q$	e	$P_\epsilon(2-e)$	P_i	$P_{0\|i}$	$P_{A\|i}$
16	2	0.00006	0.99914	0.9991	0.9952	0.9611
16	4	0.00008	1.00000	1.0000	1.0000	0.9998
32	2	0.00011	0.98157	0.9816	0.9086	0.5154
32	4	0.00013	0.99914	0.9991	0.9936	0.9421
32	8	0.00017	1.00000	1.0000	1.0000	0.9996
64	2	0.00020	0.90439	0.9042	0.6283	0.0476
64	8	0.00026	0.99914	0.9992	0.9923	0.9254
64	16	0.00034	1.00000	1.0000	1.0000	0.9994

(a) $\sigma = 0.3$

				normal		
k	$2q$	e	$P_\epsilon(2-e)$	P_i	$P_{0\|i}$	$P_{A\|i}$
16	6	0.00010	0.91672	0.9166	0.7452	0.1793
16	22	0.00026	0.99909	0.9991	0.9949	0.9589
16	40	0.00044	0.99999	1.0000	0.9999	0.9994
32	12	0.00021	0.91670	0.9171	0.6976	0.1056
32	44	0.00053	0.99909	0.9991	0.9933	0.9393
32	80	0.00089	0.99999	1.0000	0.9999	0.9989
64	22	0.00040	0.90268	0.9027	0.6238	0.0456
64	88	0.00106	0.99908	0.999	0.992	0.922
64	158	0.00176	0.99999	1.000	1.000	0.998

(b) $\sigma = 1.0$

Table 5.17: Probability of validity when $\varepsilon = 2.0$, with $\mu = 2.45$, $\varrho = 10^{-6}$, and $\lambda = 10$.

	$\sigma = 0.3$		$\sigma = 1.0$	
Distance	$\varepsilon = 1.0$	$\varepsilon = 2.0$	$\varepsilon = 1.0$	$\varepsilon = 2.0$
1	2	1	21	5
2	4	1	43	10
3	6	2	68	15
4	8	2	96	20
5	9	3	129	25
6	11	3	170	31

(a) $\varrho = 10^{-5}$ and $\lambda = 100$

	$\sigma = 0.3$		$\sigma = 1.0$	
Distance	$\varepsilon = 1.0$	$\varepsilon = 2.0$	$\varepsilon = 1.0$	$\varepsilon = 2.0$
1	2	1	20	5
2	4	1	40	10
3	6	2	59	15
4	8	2	79	20
5	9	3	98	25
6	11	3	118	30

(b) $\varrho = 10^{-6}$ and $\lambda = 10$

Table 5.18: Distance away vs. number of inquiries needed, assuming normally distributed delays with $\mu = 2.45$.

Brief mention is made in [2] of using a normal single-node inquiry and computing the mean delay from the total time, much the same way the averaging algorithm can use calculated means. For this comparison, it is assumed the algorithm in [2] takes this approach. Also, [2] shrugs off the effects of ϱ and λ. As seen in Section 5.3.5, this may not be a good idea. Since [2] specifies no means of accounting for these variables, the inherent error calculations of Section 5.3.2 are used. The net result is that a single-node inquiry of a node h hops away is much the same as a $2h$-node inquiry using the averaging algorithm. There is one difference, since the only node of interest in this inquiry is the "middle" one, N_h, the directions of inquiries do not have to be alternated, and inquiries do not have to be done in pairs. That said, Table 5.18 shows the number of inquiries needed by the algorithm in [2], assuming a probability of validity of 0.99999, for nodes from 1 to 6 hops away[5].

In the same way as done for the interval algorithm, one can compute the number of inquiries, synchronization messages, and timestamps, sent by both the single-node inquiry algorithm, and the k-node inquiry algorithm. Table 5.19 summarizes these results.

The averaging algorithm takes very well to the k-node inquiry. Whenever it is able to make estimates, it always sends fewer synchronization messages than the single-node inquiry algorithm. This is not surprising considering the nice linear growth of q with k. Because it uses the k-node inquiry, the averaging algorithm shares with the interval algorithm nice predictable synchronization message sends and receives. The single-node inquiry algorithm sends a mass confusion of messages, and in this instance there is no question that the length of time over which the messages are sent does affect the results.

5.4 Comparisons of Interval and Averaging Algorithms

Given that the preceding two sections have presented two different estimation algorithms, it is natural to wish compare the two. However, in doing so one is prone to overlook the unifying theme of this chapter — the k-node inquiry. Both algorithms use the k-node inquiry to distribute clock information, and neither modifies the inquiry in any way. Both algorithms can use the results from the same inquiry to make guesses. The two algorithms complement one another instead of competing with one another.

The examples of Sections 5.2 and 5.3 show that the interval algorithm works well when message delays are usually near the minimum, while the averaging algorithm works well

[5]If there are objections to the use of inherent error computations not found in [2], one may use the inquiry numbers from Table 5.19b since they are essentially unaffected by inherent error.

			single-node inquiry			k-node inquiry		
σ	ε	n	# inquiries	# mess	# tstamp	# inquiries	# mess	# tstamp
0.3	1.0	16	1024	2048	3072	16	496	6016
0.3	1.0	32	5088	10176	15264	30	1890	45600
0.3	1.0	64	24128	48256	72384	66	8382	403392
0.3	2.0	16	320	640	960	4	124	1504
0.3	2.0	32	1536	3072	4608	8	504	12160
0.3	2.0	64	7168	14336	21504	16	2032	97792
1.0	1.0	16	11360	22720	34080	334	10354	125584
1.0	1.0	32	58368	116736	175104	∞	∞	∞
1.0	1.0	64	288960	577920	866880	∞	∞	∞
1.0	2.0	16	2560	5120	7680	42	1302	15792
1.0	2.0	32	12800	25600	38400	86	5418	130720
1.0	2.0	64	61504	123008	184512	192	24384	1173504

(a) $\varrho = 10^{-5}$ and $\lambda = 100$

			single-node inquiry			k-node inquiry		
σ	ε	n	# inquiries	# mess	# tstamp	# inquiries	# mess	# tstamp
0.3	1.0	16	1024	2048	3072	16	496	6016
0.3	1.0	32	5088	10176	15264	30	1890	45600
0.3	1.0	64	24128	48256	72384	58	7366	354496
0.3	2.0	16	320	640	960	4	124	1504
0.3	2.0	32	1536	3072	4608	8	504	12160
0.3	2.0	64	7168	14336	21504	16	2032	97792
1.0	1.0	16	10160	20320	30480	162	5022	60912
1.0	1.0	32	50656	116736	175104	336	21168	510720
1.0	1.0	64	242624	577920	866880	728	92456	4449536
1.0	2.0	16	2560	5120	7680	40	1240	15040
1.0	2.0	32	12800	25600	38400	80	5040	121600
1.0	2.0	64	61440	122880	184320	160	20320	977920

(b) $\varrho = 10^{-6}$ and $\lambda = 10$

Table 5.19: Comparison of number of inquiries, synchronization messages, and timestamps needed in a hypercube when using single-node inquiry vs. when using k-node inquiry.

when message delays are predictable. These two conditions are not necessarily in conflict, in fact it may well be that both conditions are satisfied at the same time. If the average delay is very close to the minimum delay, both the interval and averaging algorithms may work equally well. Such a situation may be found in many modern, point-to-point systems, where each link is a dedicated high-speed line. If traffic is light, messages will not have to wait for links to clear, and delays will normally be at or near the minimum.

Even when conditions do not favor one of the algorithms, there still may be much to gain by using both. Even when the intervals for the "middle" nodes don't converge, the intervals for other nodes often will, especially the intervals for the nodes near either end. The intervals that converge can be used for their estimates, other intervals can be used as an extra check on the averaging algorithm since the actual skew must lie within the bounds of the interval, even if it doesn't converge. The averaging algorithm also can be used as a check on the interval algorithm. The estimates made by the interval algorithm should be within the uncertainties of the estimates made by the averaging algorithm, widespread disagreement between the two is an indicator of trouble.

5.5 Meshing

The probabilistic estimation algorithms presented in Sections 5.2 and 5.3 can produce estimates with any desired uncertainty, as long as k is not too large. The number of inquiries needed by interval algorithm increases rapidly with k. The number of inquiries needed by the averaging algorithm is linear with k, until the effects of inherent error are taken into account. Then, as shown in Tables 5.14 and 5.15, it may be impossible to get estimates for small uncertainties and high probabilities of validity.

In Sections 5.2.6 and 5.3.6 the single-node inquiry algorithms were able to generate acceptable estimates for all nodes when the k-node inquiries were not. They were successful because they effectively reduced k, and with it the path length. More inquiries had to be done as a result, and the result was a mass confusion of synchronization messages. What is needed is a compromise between the k-node inquiry and the single-node inquiry algorithms, one that will reduce k, but will still allow the orderly procedure of the k-node inquiry.

In this section, the k nodes are divided into a set of groups G, and each inquiry involves only the members of some element of G. Some simple algorithms for defining G such that for any N_i there will be a group in G containing N_i, and for any pair of nodes, N_i and N_j,

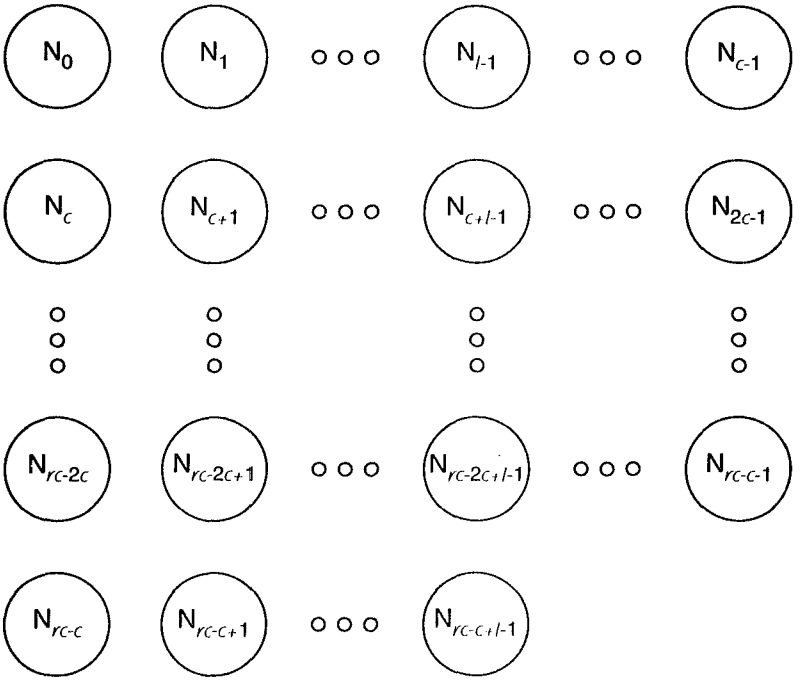

Figure 5.9: Nodes of the system laid out in a rectangular grid

there will be a group in G which contains *both* N_i and N_j. In this way every pair of nodes N_i and N_j will participate in some common inquiry, and will be able to make estimates of one another. By reducing the number of nodes involved in each inquiry, acceptable estimates may be produced which were not possible when all k nodes were involved in each inquiry. But, because the way in which the groups are defined, the procedure is regular and predictable.

These *meshing* techniques, so called because of the interlocking pattern of groups they generate, can be used to generate G for arbitrary system architectures.

5.5.1 Four-Corner Meshing

In [28] several algorithms are described to generate for each N_i a single set of nodes S_i, such that for any N_j, $S_i \cap S_j \neq \emptyset$. In particular, one way described is to lay the nodes of the system out on an $r \times c$ square grid in row major format (i.e., the node in the ith column of row j is N_{jc+i}), and to define S_i to be all nodes in the same row or column as N_i. If n is prime, or has few factors, one can choose $r \geq 3$ and $c \geq 3$ so that $n + c > rc \geq n$, and fill the excess spaces with "dummy" nodes. Figure 5.9 shows such an arrangement. If sets S_i are defined in this manner, it is easy to define the groups in G. For each pair of non-dummy nodes, N_i and N_j, where $N_j \notin S_i$, define a group consisting of N_i, N_j, and $S_i \cap S_j$.

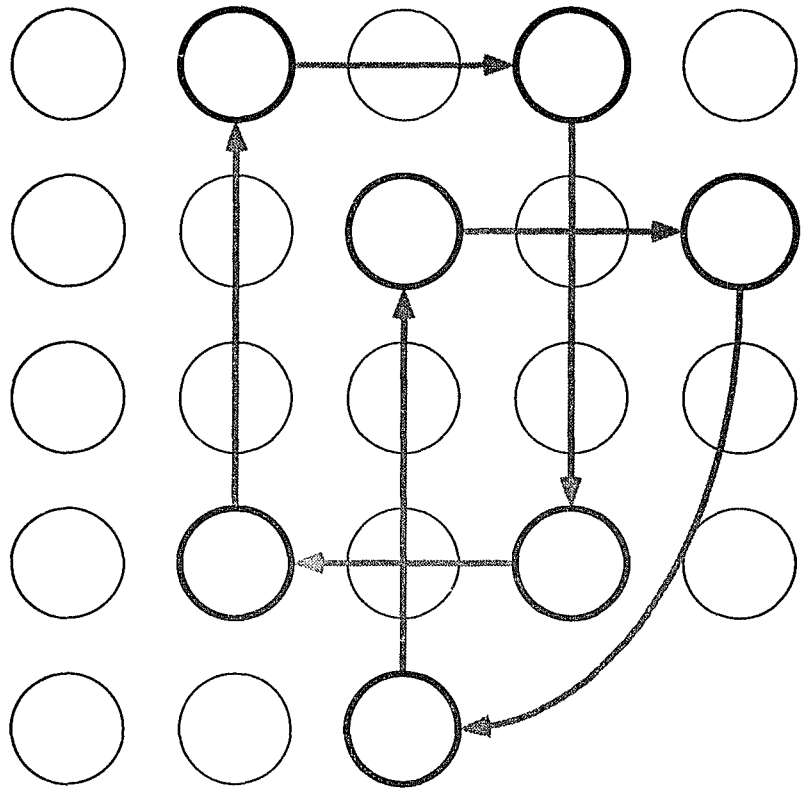

Figure 5.10: Synchronization message sends with four-corner meshing

This technique is called *four-corner meshing* because most groups will consist of four nodes which form the four corners of a rectangle in the grid. Given the group containing N_i and N_j, $N_j \notin S_i$, the inquiry will proceed with N_i sending its synchronization messages to one of the nodes in $S_i \cap S_j$, which in turn sends its synchronization messages to N_j, which sends its synchronization messages to the other node in $S_i \cap S_j$, which sends its synchronization messages to N_i. If the group contains a dummy node, the dummy node is skipped, and the synchronization message is sent to the next node instead. In case the above cycle is not the shortest available, a shorter cycle may be substituted. Figure 5.10 shows the synchronization message sends for a four node and three node group in a small system.

Number of Groups

To compute the number of synchronization messages that need to be sent one first needs to compute the number of groups that exist. Let the grid have r rows and c columns, and let there be $l \leq c$ nodes in the last row. Consider Figure 5.9 and count the groups as follows:

- Any node not in the last row of the grid will belong to $(r-2)(c-1)$ 4-node groups that don't contain a node from the last row. There are $(r-1)c$ nodes not in the last row, so there are $(r-1)c(r-2)(c-1)$ four node groups, except each group is "counted" 4 times. The total number of 4-node groups that don't contain a node from the last row is then

$$\frac{1}{4}\left(r^2c^2 - r^2c - 3rc^2 + 3rc + 2c^2 - 2c\right)$$

- Any node in the last row of the grid will belong to $(r-1)(l-1)$ 4-node groups. There are l nodes in the last row, and each group contains two nodes from the last row. The total number of 4-node groups that contain a node from the last row is then

$$\frac{1}{2}\left(rl^2 - rl - l^2 + l\right)$$

The total number of four node groups is therefore:

$$\frac{1}{4}\left(r^2c^2 - r^2c - 3rc^2 + 3rc + 2rl^2 - 2rl + 2c^2 - 2c - 2l^2 + l\right) \tag{5.45}$$

If $l < c$ there will be a number of 3-node groups in addition to the 4-node groups. The number of 3-node groups is relatively simple to calculate. For every pair of nodes, one from the last row and one from the last $c-l$ columns, there will be exactly one 3-node group containing both of them. There are $l(r-1)(c-l)$ such pairs, so there will be $l(r-1)(c-l)$ 3-node groups.

Coordination

The number of groups generated by four-corner meshing is on the order of k^2, which is the same as the number of "groups" in the single-node inquiry algorithms. It would seem therefore that four-corner meshing would fall prey to the same problems, and would flood the system with synchronization messages. A straightforward implementation would create such problems, but they can be avoided with a little careful planning.

Each node only sends synchronization messages to those nodes in the same row or column. In fact, each node sends many synchronization messages to those nodes in the same row or column, because they belong to many common groups. For example, N_0 will send a synchronization message to N_1, expecting N_1 to add its timestamp and forward the result to N_{c+1}. N_0 will send another synchronization message to N_1, expecting N_1

to add its timestamp and forward the result to N_{2c+1}. Logically, N_0 should send a single synchronization message to N_1, and have N_1 use the contents of that message when it sends synchronization messages to N_{c+1} and N_{2c+1} and the rest of the nodes in the second column.

For greatest efficiency, N_1 should wait until it receives synchronization messages from all nodes in the first row before it begins to send synchronization messages to the nodes in the second column. The process therefore requires two alternating phases: a node sends synchronization messages to each node in its row, then it sends synchronization messages to each node in its column. This requires some coordination, so that all nodes are in the same phase at approximately the same time. Since some nodes are bound to be a little slower than others, not all synchronization messages will arrive simultaneously, and arrival timestamps and node waits should be used to reduce uncertainty. A node does not include every timestamp it receives in one phase in every synchronization message it sends in the following phase. For example, N_{c+1} should not send N_0's timestamp to N_{c+2}, but it should send it to N_c. This adds some processing overhead between phases.

Coordinating four-corner meshing greatly reduces the number of synchronization messages and timestamps, N_0 will send 1 synchronization message containing 1 of its timestamps to N_1 instead of c messages each containing 1 timestamp. The reduction is on the order of $\frac{r+c}{2} - 1$ for both synchronization messages and timestamps.

Example

Consider a 64-node hypercube [38]. Each node is specified by a six-bit binary address, 000000 through 111111. Lay the nodes out in an 8×8 square grid, in row-major order, and define the groups as described above. A total of 784 4-node groups are defined.

In earlier examples it was assumed that all message delays were identically distributed. With meshing, that may not be the case. Synchronization messages may have to pass through one or more intermediate nodes in order to reach their destination. Two general cases will be considered here. The first case is that of a circuit-switched network where messages are not timestamped until the circuit is established. Message delay is then largely independent of distance, and thus the *effective* value of k is 4, at least for the purpose of determining q. The second case is that of a store-and-forward type system where the delay for a message sent a distance of h is the sum of h independent identically distributed delays. In this case it can be shown that two nodes in the same row or column will be no more than 3 hops away from one another in the hypercube. The total distance the synchronization

			interval		averaging		
eff. k	σ	ε	$2q$	$F_{\Omega_i^{2q}}(2\varepsilon)$	$2q$	e	$P_\epsilon(\varepsilon - e)$
4	0.3	1.0	4	0.99996	2	0.00112	0.99913
4	0.3	1.0	6	1.00000	4	0.00312	1.00000
4	0.3	2.0	2	1.00000	2	0.00112	1.00000
4	1.0	1.0	6	0.99988	24	0.02312	0.99929
4	1.0	1.0	8	1.00000	44	0.04312	0.99999
4	1.0	2.0	2	0.99597	6	0.00512	0.99945
4	1.0	2.0	4	0.99999	10	0.00912	0.99999
12	0.3	1.0	60	0.99903	6	0.00536	0.99909
12	0.3	1.0	102	0.99999	12	0.01136	1.00000
12	0.3	2.0	8	0.99983	2	0.00136	0.99988
12	0.3	2.0	12	1.00000	4	0.00336	1.00000
12	1.0	1.0	12	0.99972	76	0.07536	0.99900
12	1.0	1.0	16	0.99999	170	0.16936	0.99999
12	1.0	2.0	8	0.99990	18	0.01736	0.99941
12	1.0	2.0	10	0.99999	32	0.03136	0.99999

Table 5.20: Number of inquiries needed in 4 and 12 node groups when $i = k/2$, assuming normally distributed delays, with $\underset{i \to i+1}{d} = 2.11$, $\mu = 0.34$, $\varrho = 10^{-5}$, and $\lambda = 100$.

messages will have to travel in order to complete the first half of an inquiry is therefore no greater than 12, so in this case the effective value of k is 12.

Table 5.20 shows the number of inquiries needed for both 4-node and 12-node inquiries, for each estimation algorithm. Normally distributed delays are assumed, in order to allow σ to be varied. This may introduce some error in the computations for the interval algorithm, but as simulations in Section 5.2.4 showed, this error is small for small k. The delay characteristics are the same as in the first examples of Sections 5.2.5 and 5.3.5, $\varrho = 10^{-5}$, and $\lambda = 100$.

The existence of 784 groups makes it unlikely that meshing will reduce the total number of inquiries in many cases. Each group will conduct at least 2 inquiries, for at least 1568 inquiries total. The interval algorithm is a clear beneficiary of meshing. Tables 5.6 and 5.7 show that it is not possible when $k = 64$ for the interval for N_{32} to converge with a probability of 0.9 or greater. Four-corner meshing allows such systems to be successfully

synchronized. The averaging algorithm does not benefit as much, at least for the values of σ and ϵ used in the examples. Tables 5.14 and 5.15 show that when σ is large and ϵ is small the averaging algorithm cannot provide estimates with a probability of validity much greater than 0.9. Again, four-corner meshing allows such systems to be synchronized.

Simply comparing the number of inquiries does not tell the entire story. While four-corner meshing does generate more inquiries, each inquiry involves only four nodes, and so carries fewer timestamps. Using Equations (5.1) and (5.2), one finds that a single 64 node inquiry sends 127 synchronization containing a total of 6112 timestamps. A single 4 node inquiry sends only 7 synchronization messages containing a total of 22 timestamps, coordination reduces the number of synchronization messages to 1 per inquiry and the number of timestamps to 4 (8 including the arrival timestamps). However, if one wishes to compare bandwidth used one should take into account that each synchronization message in the 4 node inquiries of four-corner meshing will travel up to three hops, for effective bandwidth utilization of 3 synchronization messages and 24 timestamps (assuming coordination).

Tables 5.21 and 5.22 compare the effective number of messages and timestamps needed in the 64-node hypercube system. The tables show that while coordinated four-corner meshing usually generates more synchronization messages, it usually significantly reduces the total number of timestamps. The greatest reductions in the numbers of timestamps occur when q is large. This is due to the effects of ϱ, which become more pronounced as q increases. Also, as might be expected, the circuit switched system benefits more from meshing, primarily because the shorter message delays allow estimates to be made with fewer inquiries. The interval algorithm benefits more than the averaging algorithm, since q is non-linear with $k$in the interval algorithm, and linear with the averaging algorithm.

Another important difference between four-corner meshing and a standard k-node inquiry is run time. As can be seen in Tables 5.21 and 5.22, the number of inquiries per group is considerably less with meshing. Since all groups conduct their inquiries simultaneously, the run time for the clock distribution is the amount of time needed to conduct $2q$ inquiries in a single group. As a result, even though four-corner meshing reduces the total number of bytes sent, it also reduces the time over which they are sent, and may result in an increase in network load with respect to a k-node inquiry. On the other hand, the network load generated by the clock distribution algorithm is often a nuisance, and reducing its run time may limit its interference with other system operations.

σ	ε	$F_{\Omega_i^{2q}}(2\varepsilon)$	*eff. k*	$2q$	*eff. mess.*	*eff. tstamp*
0.3	1.0	0.99900	4	4	9408	75264
0.3	1.0	0.99900	12	60	141120	1128960
0.3	1.0	0.99900	64	∞	∞	∞
0.3	1.0	0.99999	4	6	14112	112896
0.3	1.0	0.99999	12	102	239904	1919232
0.3	1.0	0.99999	64	∞	∞	∞
0.3	2.0	0.99900	4	2	4704	37632
0.3	2.0	0.99900	12	8	18816	150528
0.3	2.0	0.99900	64	∞	∞	∞
0.3	2.0	0.99999	4	2	4704	37632
0.3	2.0	0.99999	12	12	28224	225792
0.3	2.0	0.99999	64	∞	∞	∞
1.0	1.0	0.99900	4	6	14112	112896
1.0	1.0	0.99900	12	12	28224	225792
1.0	1.0	0.99900	64	106	13462	647872
1.0	1.0	0.99999	4	8	18816	150528
1.0	1.0	0.99999	12	16	37632	301056
1.0	1.0	0.99999	64	166	21082	1014592
1.0	2.0	0.99900	4	4	9408	75264
1.0	2.0	0.99900	12	8	18816	150528
1.0	2.0	0.99900	64	72	9114	440064
1.0	2.0	0.99999	4	4	9408	75264
1.0	2.0	0.99999	12	10	23520	188160
1.0	2.0	0.99999	64	112	14224	684544

Table 5.21: Comparison of effective number of synchronization messages and timestamps when using coordinated four-corner meshing with the interval algorithm, assuming normally distributed delays, with $\underset{i \to i+1}{d} = 2.11$, $\mu = 0.34$, $\varrho = 10^{-5}$, and $\lambda = 100$.

σ	ε	$P_\epsilon(\varepsilon)$	*eff. k*	$2q$	*eff. mess.*	*eff. tstamp*
0.3	1.0	0.99900	4	2	4704	37632
0.3	1.0	0.99900	12	6	14112	112896
0.3	1.0	0.99900	64	34	4318	207808
0.3	1.0	0.99999	4	4	9408	75264
0.3	1.0	0.99999	12	12	28224	225792
0.3	1.0	0.99999	64	66	8382	403392
0.3	2.0	0.99900	4	2	4704	37632
0.3	2.0	0.99900	12	2	4704	37632
0.3	2.0	0.99900	64	8	1016	48896
0.3	2.0	0.99999	4	2	4704	37632
0.3	2.0	0.99999	12	4	9408	75264
0.3	2.0	0.99999	64	16	2032	97792
1.0	1.0	0.99900	4	24	56448	451584
1.0	1.0	0.99900	12	76	178752	1430016
1.0	1.0	0.99900	64	∞	∞	∞
1.0	1.0	0.99999	4	44	103488	827904
1.0	1.0	0.99999	12	170	399840	3198720
1.0	1.0	0.99999	64	∞	∞	∞
1.0	2.0	0.99900	4	6	14112	112896
1.0	2.0	0.99900	12	18	42336	338688
1.0	2.0	0.99900	64	96	12192	586752
1.0	2.0	0.99999	4	10	23520	188160
1.0	2.0	0.99999	12	32	75264	602112
1.0	2.0	0.99999	64	192	24384	1173504

Table 5.22: Comparison of effective number of synchronization messages and timestamps when using four-corner meshing with the averaging algorithm, assuming normally distributed delays, with $\mu = 2.45$, $\varrho = 10^{-5}$, and $\lambda = 100$.

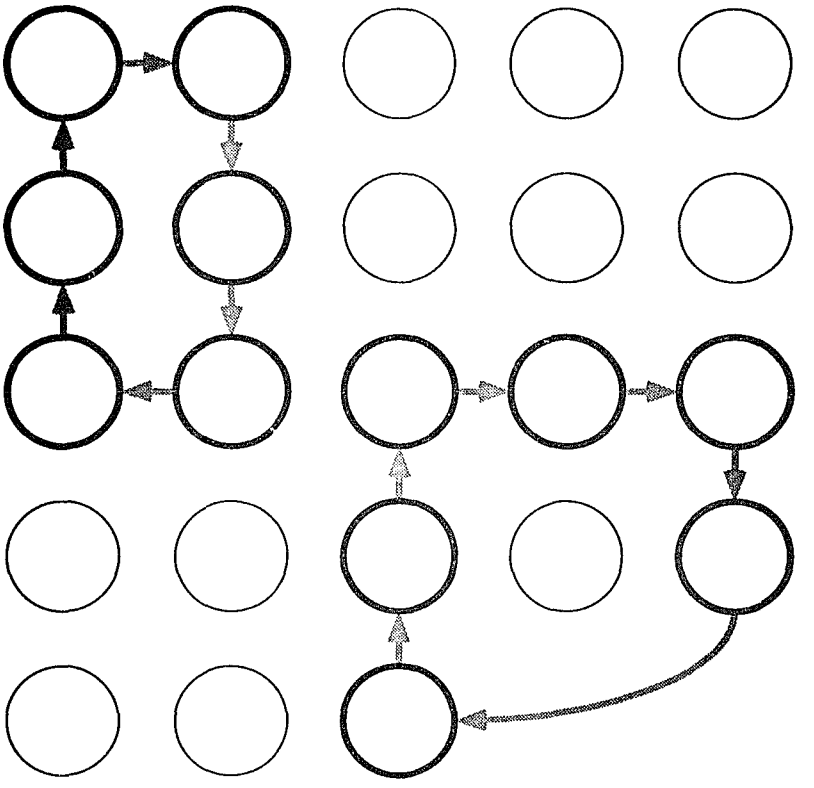

Figure 5.11: Synchronization message sends in perimeter meshing

5.5.2 Perimeter Meshing

Perimeter meshing is simply an obvious optimization of four-corner meshing, in addition to the four corner nodes each group contains the nodes on the perimeter of each rectangle. Not all of the original four-corner groups are used, only enough are used to make sure any pair of nodes share a common group. Figure 5.11 shows the synchronization message sends for two groups.

Group formation for perimeter meshing is somewhat simpler than for four-corner meshing. The nodes are laid out in an $r \times c$ rectangular grid as for four-point meshing, dummy nodes are used to fill in any blank spaces, and dummy nodes are skipped when synchronization messages are sent. However, in this case the grid is "wrapped", nodes at the top are adjacent to nodes at the bottom and nodes on the left edge are adjacent to the nodes on the right edge.

Start assigning groups with the node in row $\lfloor \frac{r}{2} \rfloor$ and column $\lfloor \frac{c}{2} \rfloor$. The process is one of drawing rectangles on the grid, where each rectangle has a corner on the node at $(\lfloor \frac{r}{2} \rfloor, \lfloor \frac{c}{2} \rfloor)$. The opposite corner of each rectangle is unique, and is used to specify it. The first four rectangles drawn will have corners at $(0,0)$, $(0,c-1)$, $(r-1,0)$, and $(r-1,c-1)$. The

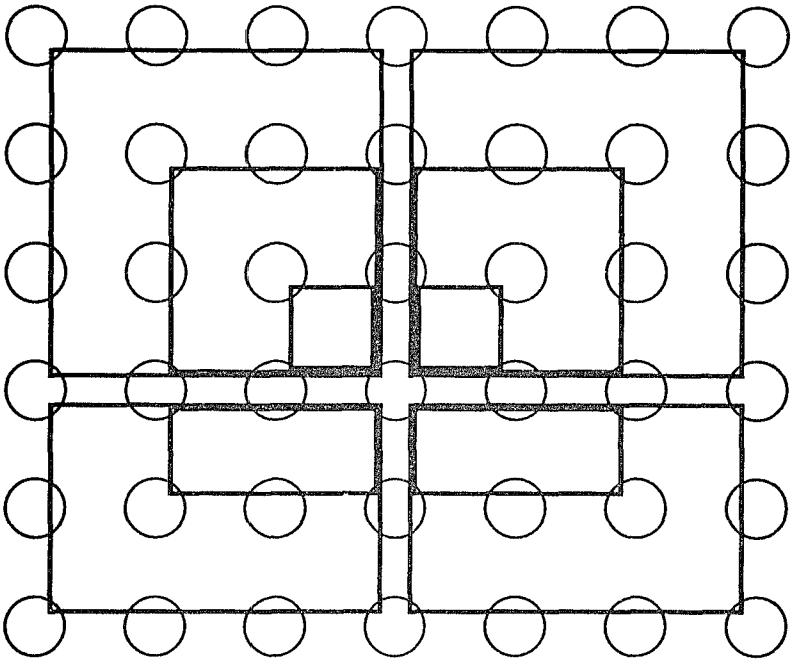

Figure 5.12: Construction of groups in perimeter meshing

next four rectangles are made by moving the corners inward along the diagonal, i.e., $(1,1)$, $(1,c-2)$, $(r-2,1)$, and $(r-2,c-2)$. This process continues until all four rectangles have either a width or height of 0. Figure 5.12 shows an example for a 7×6 mesh.

A quick look at Figure 5.12 shows that if groups are defined by the perimeters of rectangles, then there are still pairs of nodes that do not belong to a common group. To fix this the grid is "scrolled" down and to the right, nodes on the right and bottom edges wrapping around to the left and top edges. Rectangles are drawn based on the new node at location $(\lfloor \frac{r}{2} \rfloor, \lfloor \frac{c}{2} \rfloor)$, and new groups are defined. The process of scrolling the grid and drawing rectangles is repeated either $r-1$ or $c-1$ times, whichever is greater.

Perimeter meshing is an attempt to increase efficiency of four-corner meshing by allowing those nodes along the synchronization message's path to read the message and add their own timestamps, effectively adding them to the group. Many of the new, larger, groups are now redundant and can be eliminated. Perimeter meshing works well in square mesh and single-bus systems, or any system in which a square mesh may be easily embedded. However, because perimeter meshing generates groups assuming a square mesh, efficiency may suffer if the system is not a square mesh. It is possible to mimic perimeter meshing in other architectures by defining groups with four-corner meshing, then adding to each group those nodes through which its synchronization messages must pass. The problem is that it may be difficult to decide which (if any) of the new groups are now redundant and can be

eliminated.

Number of Groups

The number of groups can be computed by multiplying $\max(r, c)$ by the number of rectangles drawn each time the grid is scrolled. The number of rectangles each time the grid is scrolled can be broken down into those above and to the left, above and to the right, below and to the left, and below and to the right of $(\lfloor \frac{r}{2} \rfloor, \lfloor \frac{c}{2} \rfloor)$. The sum is then:

$$\begin{aligned} \max(r,c) \quad & \left[\min\left(\left\lfloor \frac{r}{2} \right\rfloor, \left\lfloor \frac{c}{2} \right\rfloor\right) + \min\left(\left\lfloor \frac{r}{2} \right\rfloor, \left\lceil \frac{c}{2} \right\rceil - 1\right)\right. \\ & \left. + \min\left(\left\lceil \frac{r}{2} \right\rceil - 1, \left\lfloor \frac{c}{2} \right\rfloor\right) + \min\left(\left\lceil \frac{r}{2} \right\rceil - 1, \left\lceil \frac{c}{2} \right\rceil - 1\right)\right] \end{aligned} \tag{5.46}$$

There is one caveat, when $r = c$ all the rectangles will be squares, and any square drawn below and to the right of $(\lfloor \frac{r}{2} \rfloor, \lfloor \frac{c}{2} \rfloor)$ will later be re-drawn above and to the right of $(\lfloor \frac{r}{2} \rfloor, \lfloor \frac{c}{2} \rfloor)$ after the grid has been scrolled. Some groups will therefore be counted twice. To correct for these situations, subtract r $(\lceil \frac{r}{2} \rceil)$ from the value given by Equation (5.46).

In any event, the number of groups generated by perimeter meshing should be less than $2rc$, considerably less than the number generated by four-corner meshing.

Example

Once again, consider a 64-node hypercube. The nodes are laid out in an 8×8 square grid, as before, only this time perimeter meshing is used to define the groups. Equation (5.46) gives 104 groups, but since $r = c$, 24 groups have been counted twice, so the final total is 80 different groups.

Consider the circuit-switched and store-and-forward systems used in the example for four-point meshing. The largest of the groups, the ones defined by squares with corners at $(0,0)$, $(0,4)$, $(4,0)$, and $(4,4)$, will have 16 members and requires a minimum of 18 hops for a complete cycle. In the circuit-switched system, k is effectively 16 for purposes of determining q, while in the store-and-forward system k is effectively 18. Table 5.23 shows the results, again assuming the same parameters as in Sections 5.2 and 5.3, and ϱ non-zero.

With only 80 groups, perimeter meshing still is not likely to reduce the total number of inquiries. As for four-corner meshing, perimeter meshing will only reduce the number of inquiries in extreme cases, or cases where the effects of ϱ make obtaining estimates nearly impossible. Also as with four-corner meshing, the interval algorithm is the more likely

			interval		averaging		
k	σ	ε	$2q$	$F_{\Omega_i^{2q}}(2\varepsilon)$	$2q$	e	$P_e(\varepsilon - e)$
16	0.3	1.0	∞	-	8	0.00748	0.99906
16	0.3	1.0	∞	-	16	0.01548	1.00000
16	0.3	2.0	20	0.99912	2	0.00148	0.99913
16	0.3	2.0	32	0.99999	4	0.00348	1.00000
16	1.0	1.0	14	0.99934	110	0.10948	0.99904
16	1.0	1.0	22	0.99999	∞	-	-
16	1.0	2.0	10	0.99981	24	0.02348	0.99938
16	1.0	2.0	14	1.00000	42	0.04148	0.99999
18	0.3	1.0	∞	-	10	0.00954	0.99950
18	0.3	1.0	∞	-	18	0.01754	1.00000
18	0.3	2.0	36	0.99915	4	0.00354	0.99999
18	0.3	2.0	58	0.99999	4	0.00354	0.99999
18	1.0	1.0	16	0.99943	130	0.12954	0.99906
18	1.0	1.0	24	0.99999	∞	-	-
18	1.0	2.0	10	0.99941	26	0.02554	0.99921
18	1.0	2.0	16	1.00000	46	0.04554	0.99999

Table 5.23: Number of inquiries needed in 16 and 18 node groups when $i = k/2$, assuming normally distributed delays, with $\underset{i \to i+1}{d} = 2.11$, $\mu = 0.34$, $\varrho = 10^{-5}$, and $\lambda = 100$.

beneficiary.

In comparison with four-corner meshing, perimeter meshing does what it set out to do, it reduces the number of groups and inquiries, at least in some cases. It also has some unfortunate side effects. The advantage of circuit-switched systems has disappeared. The paths are longer, so in some cases perimeter meshing cannot achieve probabilities that four-corner meshing can.

Again, the number of inquiries is not the whole story, one should consider synchronization messages and timestamps also. Determining the effective number of synchronization messages and timestamps per inquiry is not easy. Only 16 nodes participate in the inquiry, yet the total number of hops per circuit is 18. The effective number of synchronization messages and timestamps depends on where the two extra hops occur. For simplicity, it is assumed that the inquiry is identical to an 18 node inquiry, sending the same number of

σ	ε	$F_{\Omega_i^{2q}}(2\varepsilon)$	*eff. k*	$2q$	*eff. mess.*	*eff. tstamp*
0.3	1.0	0.99900	16	∞	∞	∞
0.3	1.0	0.99900	18	∞	∞	∞
0.3	1.0	0.99900	64	∞	∞	∞
0.3	1.0	0.99999	16	∞	∞	∞
0.3	1.0	0.99999	18	∞	∞	∞
0.3	1.0	0.99999	64	∞	∞	∞
0.3	2.0	0.99900	16	20	56000	763200
0.3	2.0	0.99900	18	36	100800	1373760
0.3	2.0	0.99900	64	∞	∞	∞
0.3	2.0	0.99999	16	32	89600	1221120
0.3	2.0	0.99999	18	58	162400	2213280
0.3	2.0	0.99999	64	∞	∞	∞
1.0	1.0	0.99900	16	14	39200	534240
1.0	1.0	0.99900	18	16	44800	610560
1.0	1.0	0.99900	64	106	13462	647872
1.0	1.0	0.99999	16	22	61600	839520
1.0	1.0	0.99999	18	24	67200	915840
1.0	1.0	0.99999	64	166	21082	1014592
1.0	2.0	0.99900	16	10	28000	381600
1.0	2.0	0.99900	18	10	28000	381600
1.0	2.0	0.99900	64	72	9144	440064
1.0	2.0	0.99999	16	14	39200	534240
1.0	2.0	0.99999	18	16	44800	610560
1.0	2.0	0.99999	64	112	14224	684544

Table 5.24: Comparison of effective number of synchronization messages and timestamps when using perimeter meshing with the interval algorithm, assuming normally distributed delays, with $\underset{i \to i+1}{d} = 2.11$, $\mu = 0.34$, $\varrho = 10^{-5}$, and $\lambda = 100$.

σ	ε	$P_\epsilon(\varepsilon)$	*eff. k*	$2q$	*eff. mess.*	*eff. tstamp*
0.3	1.0	0.99900	16	8	22400	305280
0.3	1.0	0.99900	18	10	28000	381600
0.3	1.0	0.99900	64	34	4318	207808
0.3	1.0	0.99999	16	16	44800	610560
0.3	1.0	0.99999	18	18	50400	686880
0.3	1.0	0.99999	64	66	8382	403392
0.3	2.0	0.99900	16	2	5600	76320
0.3	2.0	0.99900	18	4	11200	152640
0.3	2.0	0.99900	64	8	1016	48896
0.3	2.0	0.99999	16	4	11200	152640
0.3	2.0	0.99999	18	4	11200	152640
0.3	2.0	0.99999	64	16	2032	97792
1.0	1.0	0.99900	16	110	308000	4197600
1.0	1.0	0.99900	18	130	364000	4960800
1.0	1.0	0.99900	64	∞	∞	∞
1.0	2.0	0.99900	16	24	67200	915840
1.0	2.0	0.99900	18	26	72800	992160
1.0	2.0	0.99900	64	96	12192	586752
1.0	2.0	0.99999	16	42	117600	1602720
1.0	2.0	0.99999	18	46	128800	1755360
1.0	2.0	0.99999	64	192	24384	1173504

Table 5.25: Comparison of effective number of synchronization messages and timestamps when using perimeter meshing with the averaging algorithm, assuming normally distributed delays, with $\mu = 2.45$, $\varrho = 10^{-5}$, and $\lambda = 100$.

synchronization messages and timestamps. Equations (5.1) and (5.2) then give the number of synchronization messages as 35, and the number of timestamps as 477. Using these values will likely overestimate the actual numbers, but only by a small amount.

The results are in Tables 5.24 and 5.25. Compared to four-corner meshing, the savings of perimeter meshing are smaller, if even present at all. In some instances in Table 5.24, perimeter meshing with the interval algorithm cannot generate estimates with the specified uncertainty. The averaging algorithm only sees savings from perimeter meshing when estimates could not be made with a 64-node inquiry.

There is no general coordination procedure for perimeter meshing. One might try having all groups send along their "bottom" edges simultaneously, then up one side, then across the top, and so on. The problem is that while such a circuit may work well in square mesh, it may not be particularly efficient in other architectures, like the hypercube, where such a circuit may not represent the shortest cycle for the group. As a result, clock distribution with perimeter meshing is somewhat more chaotic than for an k-node inquiry or coordinated four-corner meshing. However, it is considerably better than for single-node inquiry algorithms, since the number of groups is much less.

Perimeter meshing fills a small niche between k-node inquiries and four-corner meshing. It may provide estimates with the desired uncertainty when an k-node inquiry cannot, but does not have the compressed run time of four-corner meshing. It may also be used when coordinated four-corner meshing is either not possible or undesirable.

5.5.3 Extension to Other Architectures

While four-corner meshing and perimeter meshing can be used for arbitrary architectures, they may not be particularly efficient. However, it is often possible to employ the underlying idea of four-corner meshing, only using a more natural definition of the sets for each node. That is, the set of nodes, S_i, for N_i is defined in terms of some specific feature of the architecture.

The C-wrapped hexagonal mesh [9] is one instance where the basic principles four-corner meshing map easily, producing far more efficient results than applying four-corner meshing as described above. The C-wrapped hexagonal mesh is a homogeneous structure where each node has six neighbors. Because of its homogeneity, any node can be seen as being at the center of the mesh. This is analogous to the torus, where by "scrolling" the mesh in one or more of the four available directions, one can arrange for a particular node to occupy the

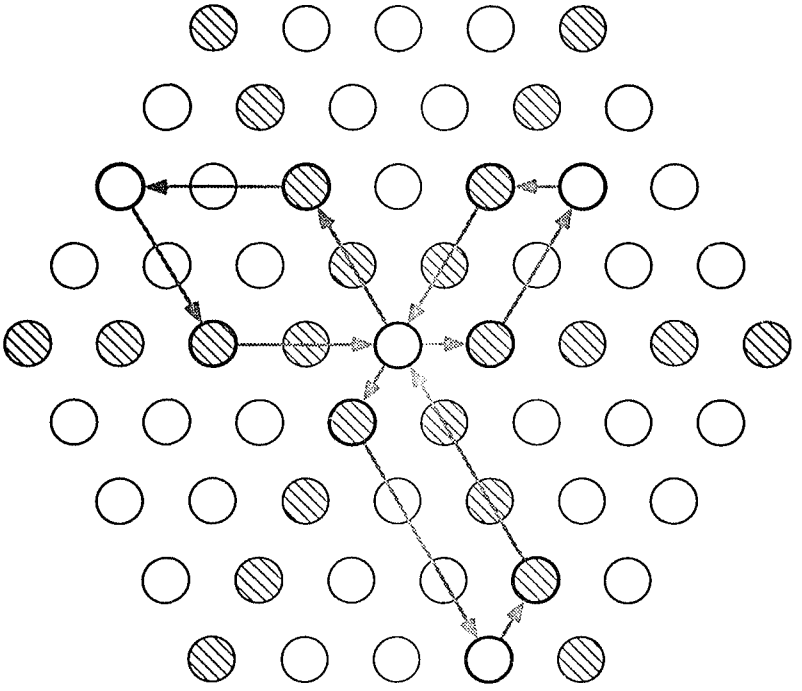

Figure 5.13: Four-corner meshing using a hexagonal mesh

center or any other specified position in the mesh. However, a C-wrapped hexagonal mesh, instead of having rows and columns, will have three major axes evenly spaced 60° apart. The set of nodes for N_i, S_i, consists of all nodes on the three axes through N_i when N_i is at the center of the mesh. Figure 5.13 shows the synchronization message sends for three different groups in a 61-node hexagonal mesh. The crosshatched nodes are those on one of the three axes of the center node. From Figure 5.13 it should also be clear that a hexagonal version of perimeter meshing is also possible.

There are a few minor points to clear up. Given N_i and N_j, $N_j \notin S_i$, $S_i \cap S_j$ will have more than two members. Groups are formed by taking the two members of $S_i \cap S_j$ closest to both N_i and N_j. Also, the six nodes at the "corners" of the hexagonal mesh will not have a common group with the center node. Six special two-node groups are therefore created, each containing the center node and one of the corner nodes.

5.6 Fault-Tolerance

The estimation algorithm is charged with the responsibility of providing accurate estimates of the local clock's skew with respect to *non-faulty* nodes. While it may be able to recognize faulty behavior, in the form of empty intervals or wildly varying guesses, it is not

required to diagnose all faults present. Dealing with estimates of faulty nodes is the domain of the clock adjustment algorithm, and Chapter 3 has already discussed this.

The estimation algorithm must, however, not allow faulty nodes to either prevent estimates of non-faulty nodes from being made, or to cause these estimates to be inaccurate. For practical purposes, the estimation algorithm cannot be expected to be completely unaffected by faults. Fault diagnosis is a hard problem in itself, and well beyond the scope of this dissertation. There are also factors beyond the control of the estimation algorithm, a faulty node may flood the network with broadcasts, effectively preventing the estimation algorithm from operating. All the same, some simple steps can be taken that will detect the presence of almost all faults, and will do so fairly quickly.

5.6.1 Chordal Messages

Because the synchronization messages are the only input to the estimation algorithm, any fault that is to affect the estimation algorithm must affect the synchronization messages. An obvious solution is to duplicate each message. Sending multiple copies of each synchronization message via independent paths will not only catch any corruption of the message en-route, but can also increase the likelihood of a near-minimal delay message [35].

However, such an approach provides no protection if the node which sends the synchronization message is faulty. In such cases it is better to avoid the faulty node altogether. *Chordal messages* are synchronization messages which are not sent to the node which would normally be the next node in the inquiry, but instead to a node that would normally be involved somewhat later. For example, a chordal message sent by N_i would not be sent to N_{i+1}, but to $N_{(i+j) \bmod k}$, where j would normally be a fairly small number.

The operation of chordal messages is straightforward. Suppose each node sends c chordal messages, then N_i will send one each to $N_{(i+2) \bmod k}$ through $N_{(i+c+1) \bmod k}$. Each recipient of a chordal message then waits for the normal synchronization message to arrive. When it arrives, its contents are checked against those of the chordal message(s). If they do not match, the chordal message may be used instead. If no synchronization message arrives before a specified timeout, the chordal message is again used instead.

Chordal messages are a powerful fault-tolerance and diagnosis tool. A faulty node which loses or alters the contents of synchronization messages will not only be detected, but the damage it caused can be undone, and it is usually easy to figure out which node is the culprit. This fault-tolerance comes at the price of extra communication overhead, each node must

send m chordal messages in order to tolerate m faults. To avoid this extra overhead in normal operation, chordal messages may only be used when a fault is suspected, and then only used long enough to locate the fault so it can be avoided.

5.6.2 Lost and Misdirected Messages

Perhaps the most obvious and easily understood fault is that of the "dead" node, which is characterized by a failure to send (or forward) a synchronization message. Closely related are those faults which cause synchronization messages to be sent or forwarded to the wrong destination. Since it may not be practical or possible for the node which receives the message to determine its actual destination, these faults are functionally equivalent to dead nodes.

Faults which lose messages are usually easy to detect and locate. In many cases the system will have already located the fault, and the estimation algorithm can use this information to avoid the faulty node. However, if the fault has occurred recently, is intermittent, or only affects synchronization messages, the system may not have located it yet, and the estimation algorithm must continue to operate in spite of the fault. Meshing is of considerable help in this regard, since a faulty node may only affect groups of which it is a member, or groups whose synchronization messages it forwards. Since the clock adjustment algorithm does not require an estimate of every other node, the loss of a few groups can be tolerated.

There are several techniques that can help prevent dead nodes from disabling a group, all involve extra communication overhead. Standard reliable communications techniques, such as acknowledgments and sending multiple copies of messages along independent paths, can detect problems caused by faulty intermediate nodes. Chordal messages can also be used, and will allow the clock distribution algorithm to continue to function until it can be re-configured to avoid the faulty node.

5.6.3 Corrupted Messages

Another common type of fault is those which alter or corrupt the contents of synchronization messages. These may be due to noisy communications lines, faulty transmitters or receivers or faults in the communications buffers. There is no way to prevent either a group member or some intermediate node from altering the contents of synchronization messages, but such tampering can be caught with the aid of digital signatures.

As for lost messages, standard reliable communications techniques may be employed

to eliminate problems caused by intermediate nodes. Since corrupted messages must be discarded, from the point of view of the estimation algorithm, corrupted messages are not much different than lost messages. The primary differences are that corrupted messages are self-evident (while lost messages are detected by timeout), and the culprit is easy to discern. If all the copies of the synchronization message N_{i+1} receives from N_i are corrupted, N_i is either responsible, or is guilty of passing on corrupted data, in either case it must be faulty. Chordal messages may still be employed to verify the identity of the guilty party, and to provide the means for bypassing it.

5.6.4 Faulty Timestamps

The final type of fault are those that cause a node to put incorrect information on a synchronization message. In contrast to faults which corrupt messages and may alter any part of it, these faults are restricted to altering the information added by the node sending the message *before* the digital signature is created. This usually means the timestamp(s) a node adds are not correct, and may be due to a faulty clock, faulty timestamping hardware, or faults which affect the operation of the estimation algorithm.

In systems where arrival timestamps are not employed, these faults are of little consequence. If the estimation algorithm succeeds in making an estimate of the faulty node (i.e., no empty intervals, or irrational or out of order timestamps occur to indicate the presence of a fault), then it becomes a problem for the clock adjustment algorithm, which has been designed to tolerate a particular number of this type of fault.

In systems where arrival timestamps are used, these faults are of greater concern. Because $W_i = T_i - T_i^a$, incorrect values for T_i and/or T_i^a will cause other nodes to compute a value for W_i which is either too high or too low. This in turn causes undetected and unaccounted for errors in the guesses, not only for N_i, but for other nodes as well. A fault at N_i which causes other nodes to incorrectly compute W_i is called an *accounting fault.*

Effects of Accounting Faults on Guesses

If the interval algorithm is used, then in the case of forward inquiries overestimating W_i will cause the lower bounds of intervals for N_{i+1} through N_{k-1} to be too high, and cause the upper bounds of intervals for N_1 through N_{i-1} to be too low. For backward inquiries, the upper bounds for N_{i+1} through N_{k-1} and the lower bounds for N_1 through N_{i-1} are affected. This can violate the guarantee that the actual skew is within the computed

interval. Underestimating W_i causes lower bounds to be too low and upper bounds to be too high. While this can slow convergence, it cannot cause the actual skew to be outside the interval, and is therefore less of a problem.

If the averaging algorithm is used, any error in W_i will directly affect the guesses for N_1 through N_{i-1} in the case of forward inquiries, or N_{i+1} through N_{k-1} in the case of backward inquiries. So while only some of the guesses will be affected, they will be affected if W_i is either underestimated or overestimated. Error in the value of W_i is essentially unaccounted-for inherent error, and decreases the probability of validity of any affected guesses and estimates.

Detecting Accounting Faults

In general, there is no sure way to detect accounting faults. There are, however, some checks that can be done and techniques that can be applied that will catch almost all of them, and limit their effects in the remaining cases. Because the algorithm in Chapter 6 uses non-zero node waits, it is susceptible to accounting faults. Therefore, a simple analysis of accounting faults is done in Section 6.3.3.

A simple sanity check that can be done is to make sure $T_i > T_i^a$. T_i^a and T_i can also be checked against T_{i-1} and T_{i+1}^a, i.e., $T_i^a > T_{i-1} + \underset{i-1 \to i}{d} - \delta$, and $T_{i+1}^a > T_i + \underset{i \to i+1}{d} - \delta$. Also, often the approximate value of W_i is known, the computed value can be compared to the expected value. Finally, if chordal messages are used, N_{i+1} can compute the expected difference between W_i and the time since the chordal message arrived, from the time between when N_{i-1} sent the chordal message and the synchronization message, and the difference in expected message delay. Occasional outliers are to be expected, but regular deviation from the expected value indicates an accounting fault is present.

The interval algorithm has a natural resistance to accounting faults. While the intervals for the "middle" nodes may be close to 2ε wide, the intervals for nearby nodes will be considerably narrower. This can be seen in Section 5.2 in the simulation results for P_{Av}, which reaches 1.0 well before the probability of convergence of the middle nodes. In general, the larger k, the narrower the intervals of nearby nodes in comparison with middle nodes. In order to avoid causing empty intervals and giving itself away, an accounting fault must limit the overestimation of its node wait to something smaller than the minimum expected width of any interval. This usually amounts to some fraction of ε, small enough that it is unlikely to be a problem, and small enough that the techniques for unreliable estimates

of Section 3.4 can be used. The interval algorithm can also help detect accounting faults that cause an underestimation of node wait. The number of intervals that converge, and the average width of intervals, should not vary much. Any sudden change in these values, such as a decrease in the number of converged intervals, or an increase in average width, indicates that an accounting fault may be present.

While the averaging algorithm by itself doesn't have much of a defense against accounting faults, it can provide an extra check on the interval algorithm. The estimates of the averaging algorithm will almost always be contained with the corresponding intervals, and will usually be smack in the middle. In order to have a significant effect on synchronization, an accounting fault must "shift" estimates either up or down. An accounting fault at N_i which causes W_i to be overestimated will shift N_0's intervals for N_1 through N_{i-1} downward, and shift N_0's intervals for N_{i+1} through N_{k-1} upward. At the same time, it will cause the averaging algorithm at N_0 to make guesses that are too high for N_{i+1} through N_{k-1}. So N_0's estimates for N_1 through N_{i-1} will be affected differently for each estimation algorithm. The resulting estimates of the averaging algorithm will be off center in their respective intervals. This not only signals the presence of the fault, but also gives an indication of its location.

5.7 Summary

Probabilistic estimation algorithms can, in theory, produce estimates with any desired uncertainty. However, they require large amounts of clock information, making each estimate expensive to produce. For this reason probabilistic estimation algorithms are usually used with master/slave adjustment algorithms. Using them with peer adjustment is far too expensive.

This chapter introduces an efficient network clock distribution algorithm, and two probabilistic estimation algorithms that work with it. The estimation algorithms are able to produce low-uncertainty estimates of every other node in moderate-sized systems at at moderate cost. The principle features of these algorithms are the following:

- Clock distribution is done in a predictable, straightforward, and orderly fashion. The system is not flooded with synchronization messages traveling every which way.
- The estimation algorithms were analyzed and simulated, and found to work well, even in larger systems.

- The clock distribution algorithm may be modified to allow for more efficient operation in large systems, or in systems where the network delay characteristics are not favorable to synchronization.
- The clock distribution algorithm is fault-tolerant, easily handling most common fault types.

CHAPTER 6

CONTINUOUS SYNCHRONIZATION

One advantage of using hardware clock distribution algorithms is the "continuous" synchronization they provide. By controlling the clock frequency at each node they make sure that clocks are brought back under control before they can stray very far. Network distribution algorithms, on the other hand, result in periodic synchronization algorithms. The algorithm runs as often as is necessary to keep the system synchronized, and is idle the rest of the time. The net result is a periodic load that can disrupt system operations, especially if a probabilistic estimation algorithm is used.

Attempting to run network distribution algorithms "continuously" (i.e., at very short intervals), either results in little gain or excessive cost. The uncertainties of absolute estimation algorithms are too high to gain much from more frequent operation, and the network traffic generated by probabilistic algorithms would consume all network bandwidth. NTP [31, 32] is continuous in the sense defined here. Estimates are made at regular intervals, allowing constant monitoring of skew, and allowing nodes to adjust their clocks as soon as an increase in skew is noticed. However, NTP uses a master/slave adjustment algorithm. No continuous synchronization algorithms which use peer adjustment are known.

The results of Chapter 5 however, provide some hope that such an algorithm may be possible. They show that a simple, efficient clock distribution algorithm can dramatically reduce the number of synchronization messages needed by probabilistic estimation algorithms. If such an algorithm can be adapted to continuous operation, it may allow probabilistic estimation algorithms to be used without overloading the system network.

This chapter describes a continuous clock distribution algorithm, and shows how it may be set up to "emulate" the clock distribution algorithm of Chapter 5. This in turn allows the probabilistic estimation algorithms of Chapter 5 to be used as well. The result is a continuous synchronization algorithm which uses probabilistic estimation, and can be used

with a peer adjustment algorithm.

The chapter starts with a discussion of the synchronization message structure. Then it considers when and where synchronization messages should be sent. The two probabilistic estimation algorithms of Chapter 5 are then adapted to continuous operation, and analyzed. A brief discussion of issues of concern to the adjustment algorithm follows. Finally, simulation is done to determine the effectiveness of the algorithm and the load it places on the system.

6.1 Synchronization Message Structure

Simply receiving a timestamp from another node does not provide enough information to make an estimate of its clock value. As was shown in Chapter 5, one must know where the timestamp has been, how long it was there and what path it took from place to place. Detailed accounting is needed to enable nodes to make accurate skew estimates.

The continuous synchronization algorithm, as will become apparent in the next few sections, depends upon accurate and extensive accounting information for each and every timestamp. Even then, long paths are out of the question, and coordinated four-corner meshing (or something similar) is required. The extra accounting information, coupled with the number of synchronization messages generated with four-corner meshing, results in a tremendous amount of synchronization information being moved through the system. To handle this load efficiently, the continuous synchronization algorithm uses a new synchronization message structure, one that allows it to combine what would otherwise be separate messages.

6.1.1 Trails

In the continuous synchronization algorithm, a synchronization message consists of a list of *trails*. Each trail contains one or more *timerecords*, listed in the order in which they were added. Each timerecord is basically an augmented timestamp and contains accounting information indicating how long the trail was at a particular node, and the minimum transmission time to the next node. Timerecords will be considered in more detail later. A typical synchronization message is shown in Figure 6.1. There is no connection between the trails in a synchronization message. They are placed together in a single message for reasons of efficiency only.

Because of the information it carries, each trail is essentially a recording of its own history, which nodes it has visited, and when. If a node receives a trail on which it already has a timerecord (i.e., the trail completes a cycle), it can use the information in the trail, along with one of the algorithms in Chapter 5, to determine approximately when the trail was at each node (according to its own clock), and compare that to the timerecords in the trail. Trails are analogous to the synchronization messages of Chapter 5, and carry essentially the same information.

Associated with each trail are several values that indicate its current state.

Length: The number of timerecords on the trail.

Arrival: The local value of $C_i^R(t)$ when the trail arrived. This is essentially the trail's arrival timestamp, except it does not include C_i^T.

Current node wait: The amount of time since the trail arrived at the current node. This is the difference between the current value of $C_i^R(t)$ and **Arrival**. Raw clock values are used to compute elapsed time because the drift of $C_i^R(t)$ is bounded by ϱ while the drift of $C_i(t)$ is not.

Trail wait: The total amount of time the trail has spent at all nodes on its path. This is the sum of the current node wait and all previous node waits.

6.1.2 Timerecords

Timerecords replace the timestamps used in conventional algorithms. Timerecords are a combination of the arrival timestamp, timestamp, and $\underset{i\to i+1}{d}$.

All timerecords have three fields. As shown in Figure 6.1, these fields are:

Arrival Time: The arrival timestamp. Only the raw clock value is actually used, the target adjustment may be dropped to save space.

Send Time: The timestamp.

Send Delay: The value of $\underset{i\to i+1}{d}$, i.e., the minimum transmission time for the trail to the node which made (or will make) the next timerecord. This information can be dropped if the value is already known, or if only the averaging algorithm is used to make estimates.

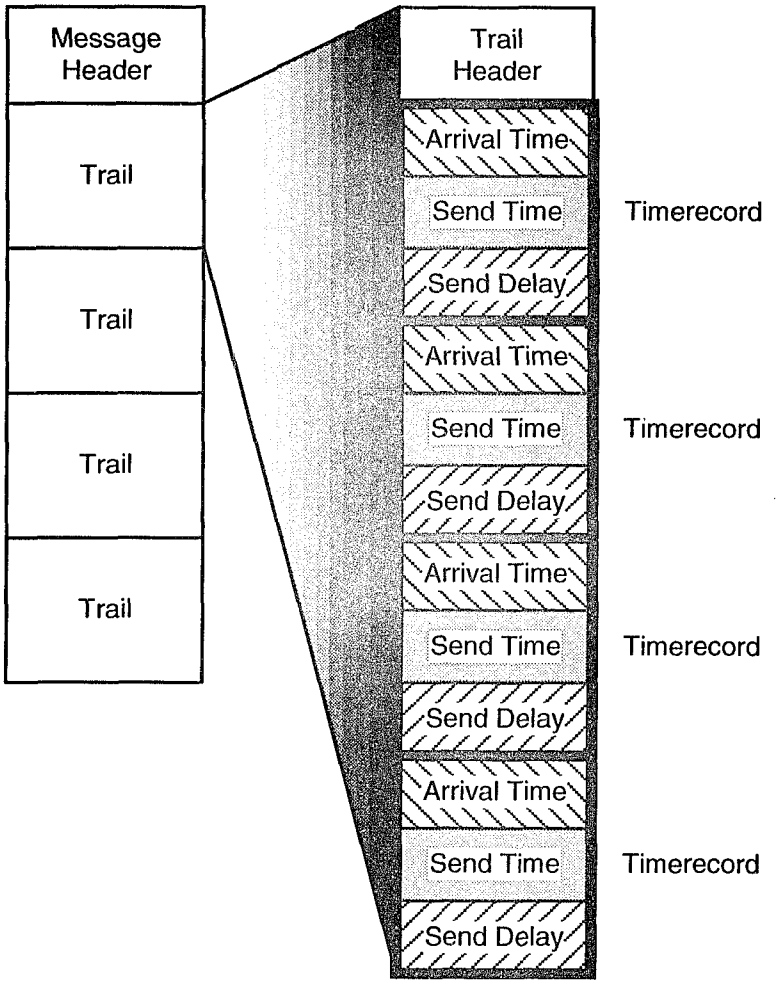

Figure 6.1: A synchronization message

The first timerecord on each trail does not have an entry in the arrival time field since the trail was created at that node. The time of creation is effectively the send time.

The more accurate the values of the arrival time and send time are, the easier and tighter the synchronization. The availability of special hardware, such as in [35], to precisely record the arrival and send times of messages can greatly improve accuracy while reducing the number of synchronization messages that need to be sent.

6.2 Message Sending

Whenever a node receives a synchronization message it breaks up the message into its component trails. Each trail is checked for cycles by checking to see if any of the timerecords on the trail were made by the local node. Those that have completed a cycle are sent to the estimation algorithm. The remaining trails are kept for a while so they can be copied

into outgoing synchronization messages.

Whenever a node sends a synchronization message, it first creates a new, empty trail and adds its timerecord to it. This is how new trails are created. The node then selects trails from those currently on hand, copies them into the synchronization message, adds its timerecord to each one, and fills in the fields appropriately. The message is then sent.

The remainder of this section presents the details of when and where messages are sent, how long a node keeps the trails it receives, and how trails are selected for outgoing synchronization messages.

6.2.1 Message Scheduling

Each N_i has a schedule for sending synchronization messages. The schedule is composed of a *round* r_i, divided into one or more *slices* $s_i^0, s_i^1, \ldots$. Each slice is a time interval along with a (possibly empty) set of nodes. After a slice's time elapses, a synchronization message is sent to each node in the slice's set, and the next slice is started. After the final slice has elapsed, the round starts over. The combination of all the rounds of all the nodes is a message sending *pattern*, that repeats over and over.

As an example, consider the wrapped square mesh in Figure 6.2. Define 2 slices for each node, each 500msec. long. After the first slice, even numbered nodes send synchronization messages to the nodes to their right and left, and odd numbered nodes send synchronization messages to the nodes above and below them. After the second slice, even numbered nodes will send synchronization messages to the nodes above and below them, and odd numbered nodes will send synchronization messages to the nodes to their right and left. If the maximum skew between clocks is δ, and $\delta \ll$ 500msec., then even nodes will be sending right and left while odd nodes are sending up and down, and vice-versa. This way no two nodes are sending synchronization messages on the same link at the same time.

Another possible message sending pattern mimics the k-node inquiry of Section 5.1.3. A cycle containing every node in the system is defined, a Hamiltonian cycle is preferable, but not necessary. Each node's round again has two slices. After the first slice each node sends a synchronization message to the next node on the cycle. After the second slice each node sends a synchronization message to the previous node on the cycle. The first slice provides the forward inquiries, the second slice provides the backward inquiries. The considerable wait time accumulated in this pattern makes it suitable for small systems only.

Yet another possible message sending pattern mimics the coordinated four-corner mesh-

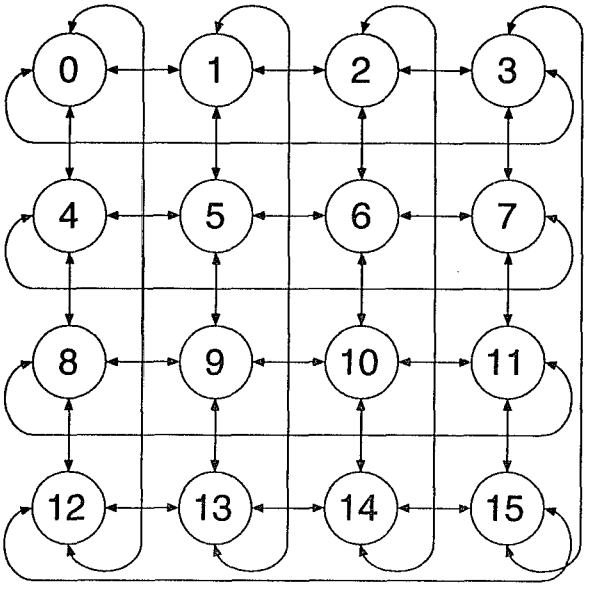

Figure 6.2: A wrapped square mesh

ing of Section 5.5.1. The system nodes are first laid out in a rectangular grid, as for four-corner meshing. The number of slices in a round is one less than the number of rows, or one less than the number of columns in the grid, whichever is larger. Two synchronization messages are sent after each slice. After the first slice each node sends one synchronization message to the node on its right, and one to the node below. After the second slice each node sends one synchronization message to the second node to its right, and one to the second node down (the grid is assumed wrapped at the edges). This continues until each node has sent one synchronization message to every other node in its row and column. As an example, in the 16-node square mesh there will be three slices per round, N_0 will send synchronization messages to N_1 and N_4 after the first slice, N_2 and N_8 after the second, and N_3 and N_{12} after the third.

6.2.2 Wait limits

The longer a trail has been at a node, the less "fresh" is the information it carries, and the less use it will be if it does manage to complete a cycle. There are two limits on the amount of time a trail may spend at any node:

W_N: The maximum node wait. The current node wait must be less than W_N.

W_T: The maximum trail wait. No trail can have a trail wait greater than W_T.

Trails which exceed either limit are discarded. A trail which is discarded without being copied into any synchronization message is said to have *expired.* An expired trail is a waste of the network bandwidth used to transmit it, and is something to be prevented.

6.2.3 Passback, Hop, and Wait Checks

It is inefficient for a node to copy all of the trails it currently has into each synchronization message. Since the goal is to form cycles, only those trails which would be able to form cycles if included in the message, should be included. There are four simple checks a node can do to decide if a particular trail should be included in a particular synchronization message:

Passback: A trail passes the passback check if either the destination of the synchronization message has no timerecords on the trail, or if it created the first timerecord on the trail. This prevents the generation of small "internal" cycles on a trail.

Minimum Hop (χ_{minh}): A trail passes the minimum hop check if the first timerecord on the trail has the property that the minimum number of timerecords the trail will acquire while traveling from the destination of the synchronization message back to the node which made the timerecord will create a cycle containing at least χ_{minh} timerecords (not including the endpoints of the cycle). This guarantees cycles of length at least χ_{minh}. Shorter cycles help produce low-uncertainty estimates, while long cycles help produce more estimates. The minimum hop check is a trade-off between uncertainty and the size of synchronization messages.

Maximum Hop (χ_{maxh}): A trail passes the maximum hop check if the first timerecord on the trail has the property that the minimum number of timerecords the trail will acquire while traveling from the destination of the synchronization message back to the node which made the timerecord will create a cycle containing no more than χ_{maxh} timerecords (not including the endpoints of the cycle). This guarantees cycles of length no more than χ_{maxh}. The number of possible cycles grows rapidly with the maximum length, while uncertainty increases. The maximum hop check is used to reduce the size of synchronization messages by eliminating trails that would be of little use to the estimation algorithm.

Wait: A trail passes the wait check if the first timerecord on the trail has the property that the minimum wait time the trail will accumulate while traveling from the destination

of the synchronization message back to the node which made the timerecord is small enough so that the trail will not exceed W_T. This guarantees that the trail will be able to form a cycle before it exceeds the maximum trail wait.

The passback check is done by checking all the timerecords on the trail. The hop and wait checks require one table lookup apiece, and can all be done at the same time. The necessary tables can be generated in n^3 time, and since their values depend only on the message schedule, they can be generated off-line. The time required to perform all the above checks on a trail is therefore a linear function of the length of the trail.

The minimum and maximum hop checks require a minimum distance table. To generate the table, create a directed graph H with a vertex for each N_i. There is an edge of length 1 from N_i to N_j if N_i sends a synchronization message to N_j. A shortest path analysis for H gives the table needed for the hop checks. The minimum hop check uses the table to make sure that that either the minimum number of hops from destination of the synchronization message to the node which created the first timerecord is large enough to ensure a cycle with χ_{minh} timerecords, or that the destination of the synchronization message is further away from the node which made the first timerecord than the current node. The maximum hop check simply uses the table to make sure the trail can return to the node which made the first timerecord before it collects too many timerecords.

The table for the wait check is more complicated to generate. If there are s slices in each round, create graph W with vertices $S_i^k, 0 \leq i < n, 0 \leq k < s$. Vertex S_i^k will represent the k'th slice of N_i. If after slice k, N_i sends a synchronization message to N_j, there is an edge from S_i^k to $S_j^{(k+1) \bmod s}$. The length of each edge will be the length of a slice. These edges are equivalent to the statement "a trail arriving at N_i at the start of slice k must wait for one slice before being sent to N_j". Furthermore, if $W_N$is ℓ times the length of a slice, S_i^k will have an edge to each member of $\{S_i^{(k+j) \bmod s} : 1 \leq j \leq \ell\}$. The length of each edge will be j times the length of a slice. These edges are equivalent to the statement "a trail arriving at N_i at the start of slice k must wait j slices for the start of slice $(k + j) \bmod s$. A shortest path analysis for W gives the table needed for wait checks. If the node which created the first timerecord on the trail is N_i, the destination of the synchronization message is N_j, and the current slice is k, the wait check looks in the table for the minimum wait from $S_j^{(k+1) \bmod s}$ to any S_i. The wait check passes if the minimum is small enough to allow the trail to complete the cycle before it expires.

6.3 Computing the Skew Estimates

The continuous clock distribution algorithm of Sections 6.1 and 6.2 is flexible enough to allow just about any estimation algorithm to be used. However, probabilistic estimation algorithms are most likely to gain from continuous clock distribution. And, since Section 6.2.1 shows how the continuous clock distribution algorithm can be made to emulate both k-node inquiries and coordinated four-corner meshing, it seems reasonable to use the probabilistic estimation algorithms of Chapter 5.

6.3.1 The Interval Algorithm

The interval algorithm adapts well to continuous operation. There is very little difference between the way it operates in Chapter 5 and the way it operates here. Each node maintains a set of intervals, one for every other node, which are updated regularly as new clock information arrives. In order to maintain these intervals, N_i must keep track of a few values for each N_j:

$I_{i,j}$: The last computed interval for N_j. This is updated every time a new timerecord from N_j arrives, or an estimate is made.

$I^W_{i,j}$: The value of $C^R_i(t_W)$, where t_W is the last time the interval was *widened*. An interval must be widened before it is used to make an estimate, or before it is updated. Widening is done to take into account any clock drift which may have occurred. To widen $I_{i,j}$, $2\varrho(C^R_i(t) - I^W_{i,j})$ is added to the upper bound of the interval and subtracted from the lower bound.

$I^R_{i,j}$: The largest raw time value seen in a timerecord from N_j. The value of $I^R_{i,j}$ will always increase.

$I^T_{i,j}$: The latest target adjustment seen in a timerecord from N_j. Whether or not a target adjustment is the latest is determined by comparing $I^R_{i,j}$ and the raw time from the timerecord.

For the sake of efficiency, $I_{i,j}$ will bound the skew between C^R_j and C^R_i. This is because $I^T_{i,j}$ and C^T_i will normally change fairly frequently. If $I_{i,j}$ bounded the skew between C^S_j and C^S_i, its endpoints would have to be updated each time either $I^T_{i,j}$ or C^T_i changed.

To make an estimate of the clock skew between N_j and N_i, N_i will first widen the old value of $I_{i,j}$. Adding the difference between current values of $I^T_{i,j}$ and C^T_i to each endpoint

of $I_{i,j}$ gives an interval containing the current skew between $C_j^S(a)$nd $C_i^S(.)$ The midpoint of this new interval is the estimated skew. The uncertainty of this estimate is half the width of $I_{i,j}$.

Newly arrived clock information is handled the same way it was in Section 5.2.1. New intervals are calculated using either Equation (5.6) or Equation (5.7), whichever is appropriate, and are then intersected with the existing intervals. The existing intervals are widened first of course, a point which is glossed over to some extent in Section 5.2.

Analysis

Analysis of the interval algorithm with continuous distribution is a straightforward application of the procedures in Section 5.2.3. The principle difference between the examples here and those in Section 5.2.5 is the need to consider the node waits generated by the continuous clock distribution algorithm. Also, exponentially distributed delays are assumed, to allow for easier comparison with the simulations of Section 6.5.

Consider a 64-node hypercube, and use the message sending pattern in Section 6.2.1 which emulates coordinated four-corner meshing. The resulting round contains 7 slices. Each inquiry involves four nodes, which are assumed to be N_0 through N_3. As pointed out in Section 5.5, there is a maximum of 3 hops between nodes.

There is a minor difficulty in that the message sending pattern being used does not send forward and backward inquiries consecutively, but rather concurrently. Inquiries of each type are started at the same time, and the time between the start of each pair is the length of a round (i.e., $\lambda$is the length of a round). A minor modification of Equations (5.17) through (5.20) fixes this: the $2\varrho\lambda$ terms are changed to $2\varrho\lambda\,(q-p-1)$.

A second minor difficulty involves the values of β_i^L and β_i^U. Examination of Equations (5.13) through (5.16) shows that β_i^L and β_i^U are different for forward and backward inquiries. When node wait is 0, the difference is small enough not to matter, and an upper bound for both values can be used. When node wait may be on the order of seconds, the difference can be very significant. The calculations in this section therefore use different values of β_i^L and β_i^U for forward and backward inquiries. Their values are computed by assuming $\underset{i\to i+1}{d} = 6.33$msec., the maximum value of $\underset{i\to i+1}{x}$ is 12msec., and the node wait at each node is one round. As an example, with a round of 1000msec., one finds that for forward inquiries $\beta_3^L = 4(6.33) + 3(12) + 3(1000) = 3061.32$, and for backward inquiries $\beta_3^L = 4(6.33) + 1(12) + 3(1000) + 2(1000) = 5037.32$.

round		N_1 and N_3		N_2	
(msec.)	q	$F_{\Omega_1^q}(2)$	$F_{\Omega_1^q}(4)$	$F_{\Omega_2^q}(2)$	$F_{\Omega_2^q}(4)$
14000	1	0.00000	0.14466	0.00000	0.00000
14000	4	0.00000	0.22351	0.00000	0.00000
14000	8	0.00000	0.22351	0.00000	0.00000
7000	1	0.00919	0.77877	0.00000	0.14441
7000	8	0.01477	0.99971	0.00000	0.45218
7000	15	0.01477	0.99974	0.00000	0.45236
3500	1	0.18630	0.93071	0.00088	0.52723
3500	15	0.76663	1.00000	0.00370	0.99923
3500	29	0.76663	1.00000	0.00370	0.99928
1750	1	0.35779	0.96553	0.01147	0.71539
1750	29	0.99953	1.00000	0.17758	1.00000
1750	58	0.99953	1.00000	0.17758	1.00000
700	1	0.46791	0.97819	0.03370	0.80347
700	72	1.00000	1.00000	0.86544	1.00000
700	143	1.00000	1.00000	0.86544	1.00000

Table 6.1: Probability of convergence using the continuous synchronization algorithm, assuming exponentially distributed delays with $\underset{i \to i+1}{d} = 2.11$, and $\mu = 0.34$.

Table 6.1 shows the results of the analysis. The round lengths are based on slice lengths of 2000, 1000, 500, 250, and 100msec. The length of a round determines λ, and therefore determines the value of q above which the probability of convergence does not increase. The probability of convergence is shown for this value of q, as well as $q = 1$, and for a q halfway in between.

The results in Table 6.1 are for the worst case, most synchronization messages will not travel 3 hops. They also ignore the fact that N_1 and N_3 are in many different groups, and will therefore be involved in many more inquiries. Even so, the results are quite good. One may recall that in Section 5.2.4 it was shown that the interval algorithm did poorly in the case of exponential delays, barely working for a 16-node system. Yet the continuous synchronization algorithm is able to get a probability of convergence of 0.999 or better for all nodes in a 64-node system when $\varepsilon = 2$msec. and slices are 1/2 second long.

6.3.2 The Averaging Algorithm

The averaging algorithm also adapts well to continuous clock distribution. Each new cyclic trail provides new guesses, which are then averaged with existing guesses. The following information is maintained by N_i for each N_j:

$A_{i,j}$: The list of most recent guesses of the skew between C_j^R and C_i^R made by N_i. The list has a limited length, and when a new guess is made, the oldest guess is removed from the list.

$A_{i,j}^R$: The largest raw time value seen in a timerecord from N_j. The value of $A_{i,j}^R$ will always increase.

$A_{i,j}^T$: The latest target adjustment seen in a timerecord from N_j. Whether or not a target adjustment is the latest is determined by comparing $A_{i,j}^R$ and the raw time from the timerecord.

Note that once again that estimates are made of the skew between C_j^R and C_i^R instead of C_j^S and C_i^S. Again, the reason is to avoid updating estimates whenever a target adjustment changes. $A_{i,j}^R$ and $A_{i,j}^T$ are the same as $I_{i,j}^R$ and $I_{i,j}^T$, and the values need only be kept once if both the interval and averaging algorithms are used.

The estimate is computed as an average of the guesses in $A_{i,j}$, plus the difference between the current values of $A_{i,j}^T$ and C_i^T. New guesses are computed using either Equation (5.36) or Equation (5.37). Old guesses are removed from $A_{i,j}$ to keep them from continuing to affect the estimates. One might recall that in Section 5.3.5 it was shown that the probability of validity actually began to decrease for large q. Removing old guesses prevents this.

Analysis

Analysis of the averaging algorithm can be done with the procedures in Sections 5.3.2 and 5.3.3. Once again, the primary difference is the node waits produced by the continuous clock distribution algorithm.

Assume the same system as with the interval algorithm. As for the interval algorithm, the fact that both forward and backward inquiries go on simultaneously requires minor modification of the analysis. In particular, the $\varrho\lambda\,(2q-1)$ terms in Equations (5.40) and (5.41) are changed to $\varrho\lambda\,(q-1)$.

round		Constant mean			Estimated mean		
(msec.)	q	e	$P_\epsilon(1-e)$	$P_\epsilon(2-e)$	e	$P_\epsilon(1-e)$	$P_\epsilon(2-e)$
14000	1	0.14000	0.85581	0.99841	0.56012	0.54491	0.98552
14000	3	0.42000	0.91197	1.00000	0.84012	0.36181	0.99935
14000	6	0.84000	0.49428	1.00000	1.26012	0.00000	0.99791
7000	1	0.07000	0.88571	0.99895	0.28102	0.77845	0.99651
7000	5	0.35000	0.98642	1.00000	0.56012	0.90513	1.00000
7000	10	0.70000	0.89281	1.00000	0.91012	0.37064	1.00000
3500	1	0.03500	0.89872	0.99915	0.14012	0.85575	0.99841
3500	10	0.35000	0.99952	1.00000	0.45512	0.99657	1.00000
3500	20	0.70000	0.97729	1.00000	0.80512	0.86111	1.00000
1750	1	0.01750	0.90476	0.99924	0.07012	0.88567	0.99895
1750	20	0.35000	1.00000	1.00000	0.40262	0.99999	1.00000
1750	40	0.70000	0.99873	1.00000	0.75262	0.99211	1.00000
700	1	0.00700	0.90824	0.99929	0.02812	0.90113	0.99919
700	50	0.35000	1.00000	1.00000	0.37112	1.00000	1.00000
700	100	0.70000	1.00000	1.00000	0.72112	1.00000	1.00000

Table 6.2: Probability of validity using the continuous synchronization algorithm, assuming exponentially distributed delays with $\underset{i \to i+1}{d} = 2.11$, and $\mu = 2.45$.

Results are summarized in Table 6.2. The same slice lengths are used as in the averaging example. The inherent error and probability of validity when $\varepsilon = 1$msec., and $\varepsilon = 2$msec., are presented for both constant means and estimated means.

For long rounds the difference between constant and estimated means is substantial. The larger inherent error for estimated means reduces the probability of validity. Also visible in the table is the point where the probability of validity begins to decrease. For example, when the round is 14000msec., the probability of validity when $\varepsilon = 1$msec. begins to drop for q larger than 3 with constant means, and for q larger than 1 with estimated means. The last item of interest is the number of guesses that must be kept in $A_{i,j}$. It should not be so large that the probability of validity is reduced. For instance, when the round is 3500msec. and $\varepsilon = 1$msec., keeping 10 guesses in $A_{i,j}$ gives a probability of validity of over 0.999, keeping 20 reduces the probability of validity to 0.977.

The results in Table 6.2 should be worst case. As for the interval algorithm, most synchronization messages will not travel 3 hops. Also, N_1 and N_3 will participate in many other

inquiries with N_0, so their probabilities of validity will be better than shown in Table 6.2. A simple modification of the above analysis could be done to compute probabilities of validity for N_1 and N_3, but this is left to the reader.

6.3.3 Accounting Faults

Because the same estimation algorithms are used, the discussion of fault-tolerance in Section 5.6 applies equally well here. However, since the continuous clock distribution algorithm generates node waits, the estimation algorithms must now worry about accounting faults.

There is no sure way to detect accounting faults, but that is largely due to the fact that their effects may be very slight. The interval algorithm will often detect accounting faults by generating empty intervals, but it is not guaranteed to detect them. It is possible to modify the analysis of Section 5.2.3 to incorporate a simple model of accounting faults. It is then possible to determine the probability of empty intervals, as well as the probability that the actual skew lies outside the interval.

For this analysis an accounting fault is modeled as an offset added to the node wait of the faulty node. One offset, o_f, is added to the node wait during forward inquiries, and a second offset, o_b, is added to the node wait during backward inquiries. The values of o_f and o_b may or may not be equal, and may be either positive or negative.

Assume that the accounting fault is at N_j, and consider the intervals computed for N_i, where $i > j$. The lower bound for N_i's interval generated by backward inquiries will be o_b too high. The upper bound for N_i's interval generated by forward inquiries will be o_f too low. New distribution functions for the lower and upper bounds may be derived to replace those in Equations (5.17) and (5.19).

$$
\begin{aligned}
F'_{\mathcal{L}_i^p}(x) &= \left(1 - F_{\underset{i\to k}{\mathcal{X}}}\left(\frac{-x - 2\varrho\beta_i^L + o_b - 2\varrho\lambda(q-p-1)}{1+\varrho}\right)\right) \\
&\qquad \times \left(1 - F_{\underset{i\to 0}{\mathcal{X}}}\left(\frac{-x - 2\varrho\beta_i^L - 2\varrho\lambda(q-p-1)}{1+\varrho}\right)\right) \qquad (6.1) \\
F_{\mathcal{U}_i^p}(x) &= 1 - \left(1 - F_{\underset{0\to i}{\mathcal{X}}}\left(\frac{x - 2\varrho\beta_i^U + o_f - 2\varrho\lambda(q-p-1)}{1+\varrho}\right)\right) \\
&\qquad \times \left(1 - F_{\underset{k\to i}{\mathcal{X}}}\left(\frac{x - 2\varrho\beta_i^U - 2\varrho\lambda(q-p-1)}{1+\varrho}\right)\right) \qquad (6.2)
\end{aligned}
$$

The density functions can be found through differentiation. New distribution and

o	round (msec.)	q	N_2				N_3			
			$P_{E_2^{2q}}$	$P_{\mathcal{L}_2^{2q}}$	$P_{\mathcal{U}_2^{2q}}$	$P_{C_2^{2q}}$	$P_{E_3^{2q}}$	$P_{\mathcal{L}_3^{2q}}$	$P_{\mathcal{U}_3^{2q}}$	$P_{C_3^{2q}}$
0.5	7000	15	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.00000	1.00000
0.5	3500	29	0.00000	0.00007	0.00000	0.99997	0.00000	0.00000	0.00000	1.00000
0.5	1750	58	0.00000	0.00173	0.00008	0.99819	0.00000	0.00000	0.00000	1.00000
0.5	700	143	0.00031	0.01292	0.00556	0.98160	0.00000	0.00003	0.00001	0.99997
1.0	7000	15	0.00000	0.00248	0.00000	0.99752	0.00000	0.00000	0.00000	1.00000
1.0	3500	29	0.00096	0.04969	0.00336	0.94711	0.00000	0.00028	0.00000	0.99972
1.0	1750	58	0.05923	0.20602	0.08482	0.72664	0.00001	0.00317	0.00069	0.99615
1.0	700	143	0.54748	0.58481	0.46859	0.22063	0.00135	0.01839	0.01110	0.97071
2.0	7000	15	0.18738	0.55600	0.08130	0.40791	0.00014	0.03041	0.00053	0.96908
2.0	3500	29	0.95865	0.95661	0.76707	0.01011	0.05768	0.22233	0.07112	0.72236
2.0	1750	58	1.00000	0.99976	0.99748	0.00000	0.51371	0.58732	0.41369	0.24196
2.0	700	143	1.00000	1.00000	1.00000	0.00000	0.98555	0.95044	0.91758	0.00408

Table 6.3: Analysis of accounting faults when using the continuous synchronization algorithm, assuming exponentially distributed delays with $d_{i \to i+1} = 2.11$, and $\mu = 2.45$.

density functions for MAX_i^{2q} and MIN_i^{2q} can be found by modifying Equations (5.21) through (5.24) to use the new functions for the lower and upper bounds. The probability of an empty interval, $P_{E_i^{2q}}$, is then the probability that $MIN_i^{2q} - MAX_i^{2q} < 0$. This can be found with a simple convolution integral, much like the one in Equation (5.26):

$$\begin{aligned} P_{E_i^{2q}} &= \int_{-\infty}^{\infty} dy \int_{-\infty}^{y} f'_{MAX_i^{2q}}(y) f'_{MIN_i^{2q}}(z) dz \\ &= \int_{-\infty}^{\infty} f'_{MAX_i^{2q}}(y) F'_{MIN_i^{2q}}(y) dy \end{aligned} \tag{6.3}$$

Furthermore, the probability that the actual skew is less than the lower bound of the interval is $P_{\mathcal{L}_i^{2q}} = 1 - F'_{MAX_i^{2q}}(0)$, and the probability that the actual skew is greater than the upper bound of the interval is $P_{\mathcal{U}_i^{2q}} = F'_{MIN_i^{2q}}(0)$. The probability that the actual skew is within the interval is $P_{C_i^{2q}} = (1 - P_{\mathcal{L}_i^{2q}})(1 - P_{\mathcal{U}_i^{2q}})$.

As an example, consider the same system used in the examples above. Assume that N_1 has an accounting fault, and for simplicity, $o_f = o_b = o$. N_0's intervals for N_2 and N_3 will be affected, the above equations determine how likely it is that the intervals will become empty, and how likely it is that the actual skew no longer lies within the interval.

Table 6.3 summarizes the results. Several different values of o are used. For each round

length, q was chosen to give a maximum probability of convergence, so the intervals would be at their smallest.

The general trend of the results is that accounting faults have little effect when o is small in comparison to the expected interval width. Longer rounds have no trouble when $o = 0.5$msec., but then their expected interval width is greater than 4msec. Even for rounds of 500msec. the interval for N_2 almost always contains the actual skew. Empty intervals are not a likely occurrence until it is quite likely that the interval does not contain the actual skew value. They do not do a good job of catching a small o, but they do manage to catch larger ones.

Somewhat surprising at first glance is the effect of the accounting fault on the interval for N_3. From Table 6.1 it is clear that the interval for N_3 is almost always smaller than the interval for N_2. But even with a large o it almost always contains the actual skew value and almost never becomes empty. The reason is that the accounting fault affects the upper bound of N_3's interval on forward inquiries, and the lower bound on backward inquiries. A look at Figure 5.3 shows that the upper bound for N_3 computed on forward inquiries will be too high. Similarly, on backward inquiries the lower bound computed for N_3 will be too low. The accounting fault is therefore affecting bounds that are already bad, and so has little effect. What this means is that an accounting fault can only affect estimates of nodes in the same row or column, e.g., the accounting fault at N_1 will affect at most 7 (out of 63) of N_0's estimates. The effects will also be small, less than ε(otherwise empty intervals will result), so the methods in Section 3.4 may be used to deal with them.

6.4 Uncoordinated Adjustment

Continuous synchronization can be done with just about any adjustment algorithm. The algorithms of Chapters 3 and 4 both work well. As do interactive convergence [22], and interactive consistency [22][1]. Even master/slave adjustment algorithms or hierarchical adjustment algorithms can be used[2].

All the above adjustment algorithms assume that all nodes adjust their clocks simultaneously, usually as soon as the estimates become available. With continuous synchronization the estimates are always available, and there is no obvious time to compute the synchroniza-

[1] With the proviso that Byzantine behavior is not allowed

[2] Researchers at Martin Marietta are considering using the continuous synchronization algorithm in a master/slave setting. The continuous availability of skew estimates allows more precise coordination of events than synchronization alone.

tion adjustment. *Coordinated adjustment* solves this problem by specifying certain times when all nodes adjust their clocks, making clock adjustment roughly simultaneous. Because estimates are continuously available, the adjustments can take place more often, allowing for larger values for τ, smaller values for δ, or both. However, this approach violates the "spirit" of continuous synchronization, it is also rather inflexible. Nodes must adjust their clocks at the specified times. And if they cannot, either because they are busy, or because estimate uncertainty is too high, they risk falling out of synchronization.

Uncoordinated adjustment allows nodes to adjust their clocks just about whenever they wish. Nodes may therefore wait until they are not busy, or until estimate uncertainty is low enough. The principle hazard is that nodes may wait too long.

With uncoordinated adjustment each node normally attempts to compute a synchronization adjustment either at regular intervals, or whenever it has the time. The computation is said to be *successful* if all the conditions of the adjustment algorithm (e.g., minimum number of accepted estimates, or average uncertainty) are met and a synchronization adjustment is produced. If the absolute value of the synchronization adjustment is greater than a specified minimum, the synchronization adjustment is *applied* by adding it to the target adjustment.

There are three reasons for a minimum size of the synchronization adjustment: estimation uncertainty, faulty nodes, and estimate bias. When a node adjusts its clock it should be sure that the adjustment reflects a real change in skew, and not just variation in the estimates. The estimates made by the continuous synchronization algorithm will normally "jump" around some center point, and thus the computed synchronization adjustment will change slightly from moment to moment. The minimum synchronization adjustment should therefore be at least ε. Faulty nodes can also cause the synchronization adjustment to suddenly change. Worse, if all the faulty nodes read either fast or slow, the non-faulty nodes may start to "chase" them, constantly increasing or decreasing their clocks. The minimum synchronization adjustment should therefore be greater than the maximum effect of faulty nodes on the synchronization adjustment (e.g., $m\delta/\zeta$ for the algorithms in Chapter 3. Finally, estimation algorithms may have an inherent bias. The interval algorithm, for instance, tends to overestimate skew. Synchronization adjustments therefore tend to be slightly larger than they should be, and as a result clocks will tend to run fast on average. A minimum synchronization adjustment cannot prevent this from happening, but it can slow it down by reducing the number of times a synchronization adjustment is applied.

There is still the question of proving that the system will remain in synchronization. Unfortunately, uncoordinated adjustment does not provide any means of doing this. That the system remains in synchronization is normally shown through analysis and testing. However, some comfort can be derived from the availability of estimates. If ε is small in comparison to δ, nodes will always have a very good idea of just how well synchronized they are. So if the system does fall out of synchronization, or is in danger of doing so, the nodes will be aware of it and may take steps to counteract it.

6.5 Simulation

As in Chapter 5 simulation is used to demonstrate the effectiveness of synchronization using continuous clock distribution. Unlike Chapter 5, the simulations here are of an operating system. They are designed to evaluate the performance of the continuous synchronization algorithm under "real-life" circumstances, not just verify that it works. The goal of simulation is not so much to see if the system remains synchronized, but to determine how much of a load it generates.

All simulation is done using the interval estimation algorithm, and the adjustment algorithms using reliable estimates of Section 3.3.

6.5.1 Simulation Parameters

In order to accurately simulate system operation, a number of system parameters must be specified. These parameters are summarized in Table 6.4.

The message delays are the same as those used in Chapter 5. Message transmission is done by store-and-forward, and individual hops are assumed to be independent and identically distributed, with an exponential distribution. Thus, the delay for a 3-hop path is at least 6330μsec., and averages 7350μsec.

The length of the various components of synchronization messages are specified so that the total number of bytes sent by the clock distribution algorithm may be determined. The values used are the lengths of the corresponding structures in the simulator.

The values of ε, δ, W_N, W_T, and the length of a slice are varied from example to example. Unlike the simulator of Chapter 5, which simulated a single run of the synchronization algorithm, the simulator used here operates for a specified period of time, covering multiple clock adjustments by each node. The simulator was set to run for several hours of

Parameter	Value
Maximum clock drift (ϱ)	0.00001
Minimum sending time ($\underset{i \to i+1}{d}$)	2110μsec.
Mean sending time ($\underset{i \to i+1}{\mu}$)	2450μsec.
Length of timerecord	32 bytes
Length of trail header	24 bytes
Length of message header	16 bytes

Table 6.4: Simulation Parameters

simulated time in order to make sure there were no slips or breakdowns on the part of the synchronization algorithm.

6.5.2 16-node Square Mesh

The first example is the 16-node square mesh described in Section 6.2.1, using the first message sending pattern (where nodes send only to neighboring nodes). It is assumed no faults are present ($m = 0$), $\varepsilon = 1$msec., and $\delta = 5$msec. Using range restriction, and Equation (3.10), one finds $\zeta > 10$. Start with a slice length of $\frac{1}{2}$sec., $W_N = 1$sec., and no hop checking. Under these circumstances the system synchronizes, averaging about 13 estimates per successful computation. The average synchronization message is about 3.5K bytes long. Each link averages one synchronization message per second, so given a 10Mbit/sec. link, only about 0.3% of the available bandwidth is used for synchronization. Changing the minimum hop check to 3 and the maximum hop check to 5 decreased the average size of a synchronization message to about 3.3K bytes. Increasing the minimum hop check any further increases estimate uncertainty to the point where it is difficult to successfully compute a synchronization adjustment.

The synchronization message sending pattern in the previous example is not particularly efficient. Nodes send only to their neighbors, so in order to make an estimate of a node h hops away a trail will have to make at least $2h$ hops, accruing considerable trail wait in the process. For the next example the last message sending pattern described in Section 6.2.1, the one which emulates coordinated four-corner meshing, is used instead. Use the same values as above for m, ε, δ, and the length of the slice. Let W_N be 1 round, W_T be three rounds, and the minimum and maximum hop checks both be 3. Under these circumstances

the system synchronizes, averaging nearly 16 estimates per successful computation. The synchronization messages average 1.8K bytes. The average number of messages per link is now 2/sec., so synchronization again uses about 0.3% of the available bandwidth of a 10Mbit/sec. link. Only in this case the synchronization is much tighter because of the greater number of estimates used when computing each synchronization adjustment.

6.5.3 64-node Hypercube

A 64-node hypercube [38] poses a much greater test for the continuous synchronization algorithm. Again, the message sending pattern which emulates coordinated four-corner meshing is used. This produces a round that is seven slices long. W_N is 1 round, W_T is 3 rounds, and the hop checks are both 3. The resulting synchronization messages are about 12K bytes long, independent of slice length. These messages are probably too long to be sent at once, but could easily be broken up into multiple messages and sent separately over the course of the next slice, as long as no trail is split between two messages.

For the first example, range restricted mean is used to compute the synchronization adjustment. Table 6.5 summarizes the results. Each node attempts to compute a synchronization adjustment at the end of each round, the sixth column shows the average number of estimates used in each successful attempt. The last column shows if the simulation is able to synchronize the system. The simulation made no attempt to simulate faults, only increased ζ accordingly.

One can get a good idea of how easily the system is synchronized by comparing $\zeta$and the average number of estimates per successful computation of the synchronization adjustment. If the average number of estimates is little more than ζ, the system had a difficult time staying synchronized. A real system, with real faults, might not be so lucky. The effects of system parameters on synchronization is what would be expected. Reducing δ or ε, or increasing m, makes synchronization more difficult. Reducing the slice time reduces the average uncertainty in the estimates, and makes more estimates available. So, for example, if $m = 4$, and $\delta = 8$msec., synchronization may be achieved with $\varepsilon = 2$msec., and a slice time of 1 second. However, if δ is reduced to 6msec., the slice time must be reduced to $\frac{1}{2}$ second.

To compute bandwidth used, note that when slices are 1 second long each node will generate 2 messages per second, spread over 6 links. Since synchronization messages averaged 12K bytes, the resulting bandwidth usage is about 0.64%. For $\frac{1}{2}$ second slices the

ε (msec)	δ (msec)	m	slice (msec)	ζ	estimates	synchronized?
2	10	0	1000	41	49	y
2	10	12	500	56	62	y
2	8	0	1000	43	49	y
2	8	4	1000	49	52	y
2	8	0	500	43	61	y
2	8	4	500	49	62	y
2	8	0	250	43	63	y
2	8	4	250	49	64	y
2	8	8	250	54	64	y
2	8	12	250	59	64	y
2	6	0	1000	49	51	y
2	6	4	1000	55	55	n
2	6	0	500	49	61	y
2	6	4	500	55	62	y
2	6	0	250	49	63	y
2	6	4	250	55	64	y
2	6	8	250	61	64	y
2	5	0	250	54	64	y
1	5	0	500	41	-	n
1	5	0	250	41	41	y

Table 6.5: Simulation results for a 6-cube with range restriction

ε (msec)	δ (msec)	m	slice (msec)	ζ	estimates	synchronized?
2	8	0	1000	43	49	y
2	8	4	1000	51	53	y
2	8	0	500	43	61	y
2	8	4	500	51	62	y
2	8	0	250	43	63	y
2	8	4	250	51	64	y
2	6	0	1000	49	51	y
2	6	4	1000	58	58	n
2	6	0	500	49	60	y
2	6	4	500	58	62	y
2	6	0	250	49	63	y

Table 6.6: Simulation results for a 6-cube with unrestricted range mean.

bandwidth usage doubles, to about 1.3%. For $\frac{1}{4}$ second slices the bandwidth usage doubles again, to about 2.6%. Consulting Table 6.5, it can be seen that for only 1.3% of the network bandwidth the system can be synchronized to within 6msec., and still tolerate 4 faults.

Simulation were also done with unrestricted range mean. Table 6.6 summarizes the results. They are much the same as for the range restricted case, only the value of ζ increases when $m \neq 0$. The safety margin that was present with range restriction has been cut somewhat.

6.5.4 A Real-World Simulation

To further prove the viability of the continuous synchronization algorithm a second simulator was constructed which attempts to synchronize the clocks of a group of Sun workstations. The simulator consists of several *synchronization processes* running on various workstations. Each synchronization process reads the local workstation clock, sends and receives synchronization messages using Internet datagrams, and uses the synchronization algorithm to compute its own target adjustment. Since each process maintains a target adjustment instead of altering the local clock, more than one synchronization process can run on each workstation.

Each synchronization process was configured to act as though it were the synchronization

algorithm of a particular node in a hypercube. Each sent and received synchronization messages only from its "neighbors" in the hypercube. Eight workstations were used, running two synchronization processes each, to simulate a 16-node hypercube. The workstations were spread over two separate Ethernets, connected by an FDDI bridge. The synchronization processes were arranged so that the pair on each workstation simulated nodes on the opposite ends of the hypercube (e.g., N_0 and N_{15}), so the two synchronization processes would never communicate directly. In fact, the two synchronization processes on each workstation should have the most trouble estimating each other's skews.

The slice interval was set at 2 seconds. Each simulation process measured the minimum delay to each of its neighbors, minimum times were about 1200μsec. between workstations on the same Ethernet and about 1500μsec. when the workstations were in different Ethernets. The sending pattern was derived the same way as for the 6-cube examples above, resulting in three slices per round. Test runs found a ϱ smaller than 0.00005 caused errors. The synchronization processes were set up to use range restricted mean with ε = 5msec., δ = 20msec. They were run for two hours during the middle of the afternoon when workstation usage and network traffic was fairly high. Each synchronization process generated a report at regular intervals, and included in each report is the current target adjustment. The adjustments of two synchronization processes running on the same workstation should be the same, comparing the reports of the two processes shows how well the synchronization algorithm worked.

Figures 6.3 through 6.10 plot the target adjustments of each pair of nodes in the simulation. It is obvious from the graphs that while the target adjustments of simulation processes running on the same workstation are nearly identical, there is considerable difference in target adjustments between simulation processes running on different workstations. Since simulation processes on the same workstation use the same raw clock value, they should have the same target adjustment if the synchronization algorithm works. Obviously, the synchronization algorithm did work, on each graph the curves are nearly indistinguishable, differing by no more than 10msec.

Another obvious feature of the graphs is the positive slope exhibited in all but Figure 6.7. This is largely due to the estimation bias of the interval estimation algorithm. Evidently the workstation which was running simulation processes 4 and 11 had a fast running clock. As said in Section 6.4, there is nothing that can be done to stop the effects of estimate bias, it can only be slowed down. On average, the graphs show an increase in the target

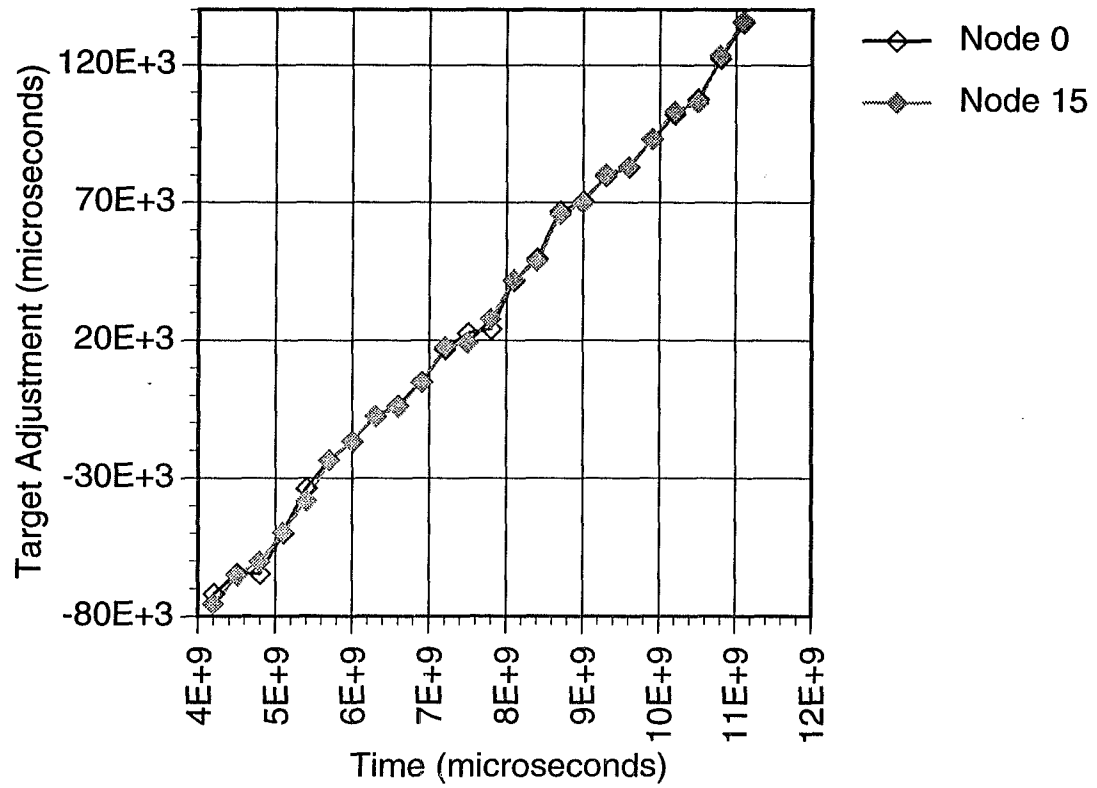

Figure 6.3: Clock adjustments of N_0 and N_{15}

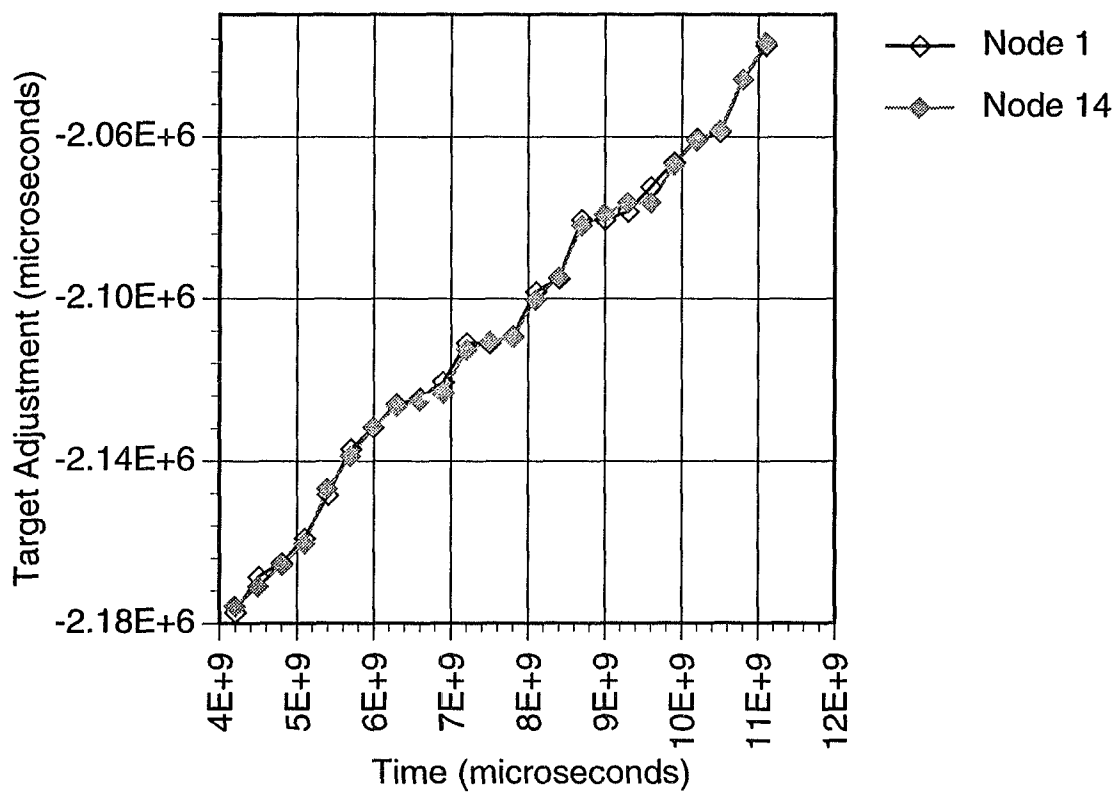

Figure 6.4: Clock adjustments of N_1 and N_{14}

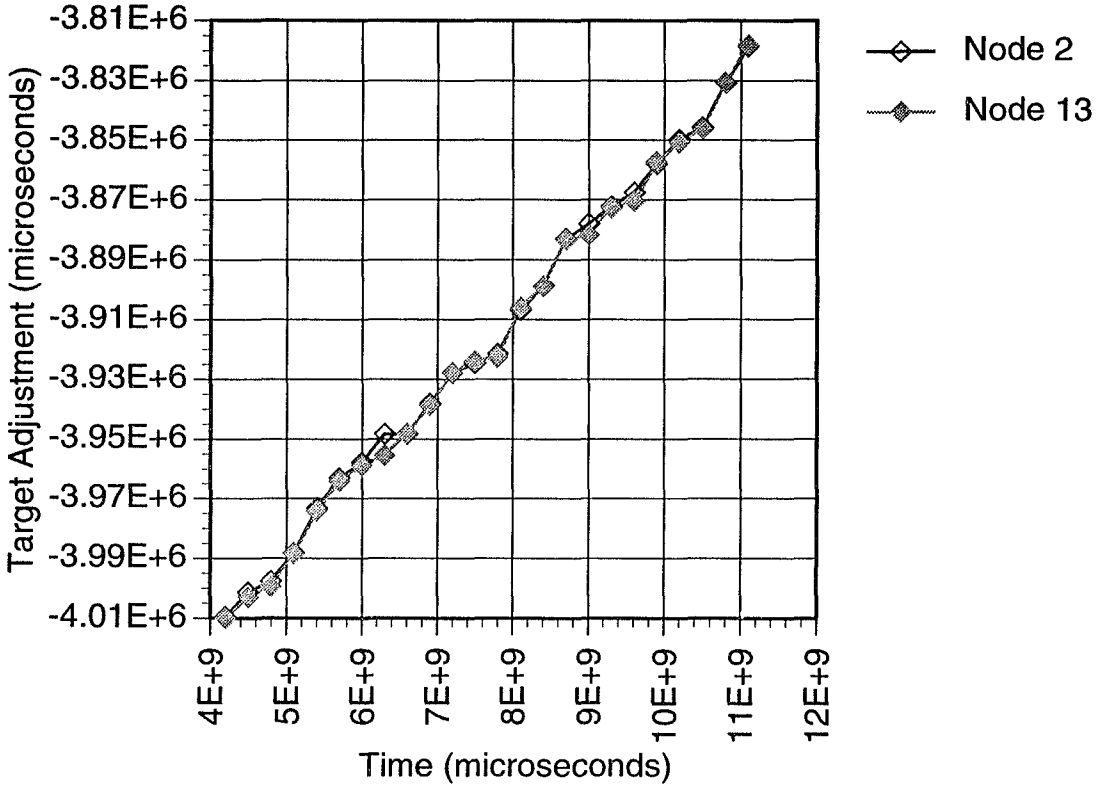

Figure 6.5: Clock adjustments of N_2 and N_{13}

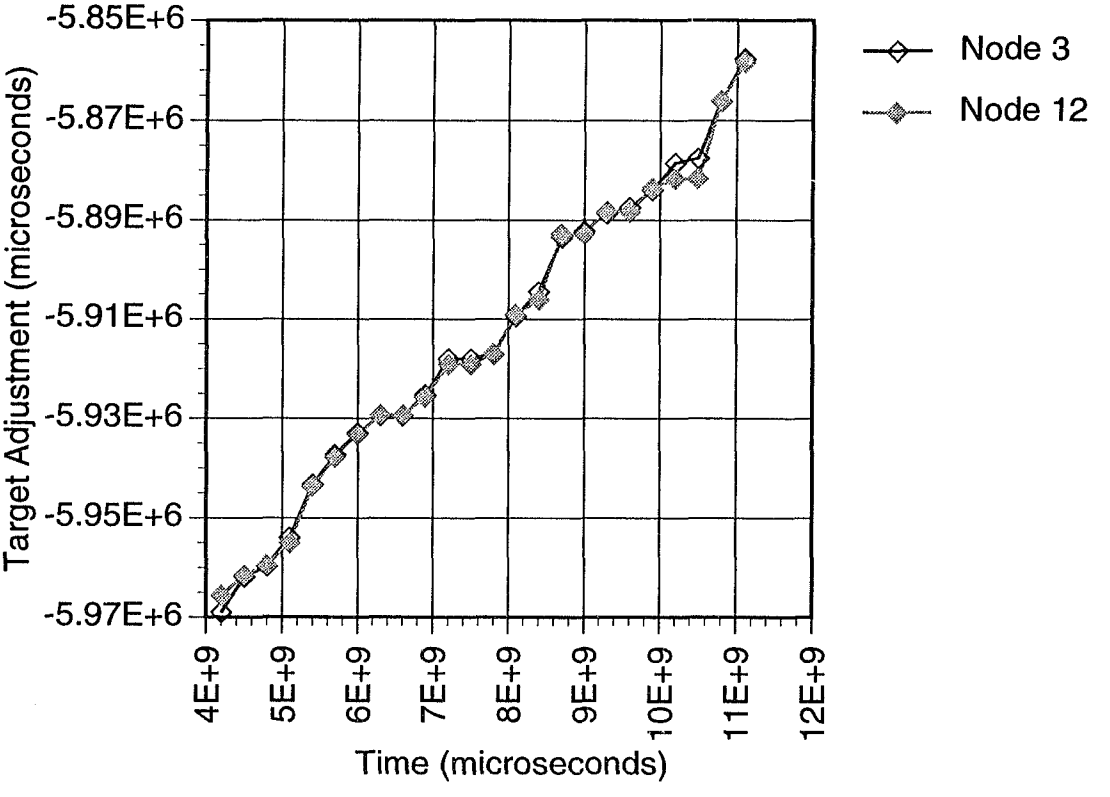

Figure 6.6: Clock adjustments of N_3 and N_{12}

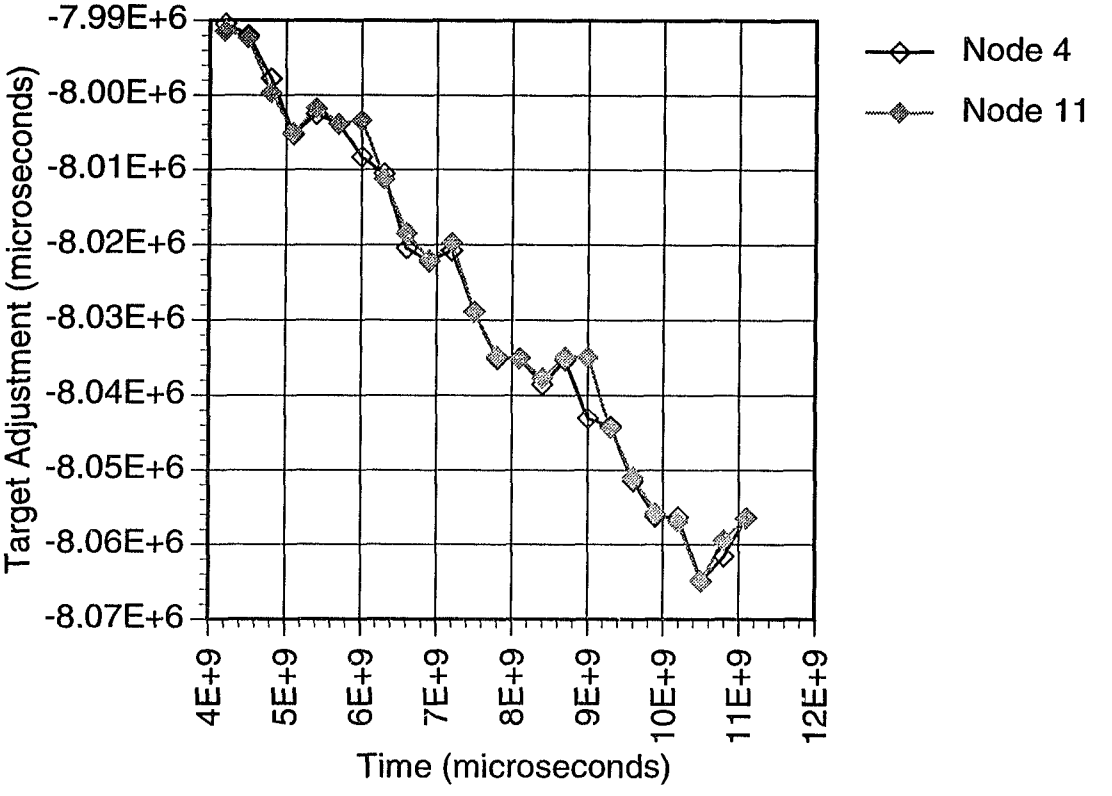

Figure 6.7: Clock adjustments of N_4 and N_{11}

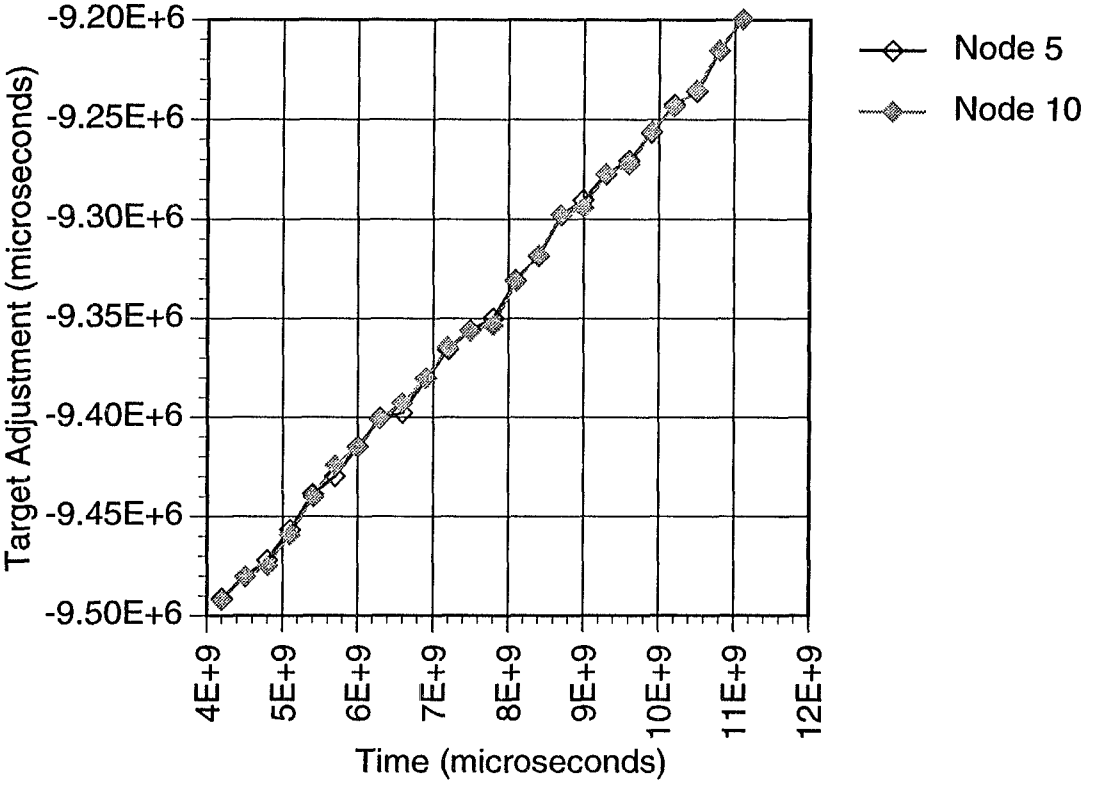

Figure 6.8: Clock adjustments of N_5 and N_{10}

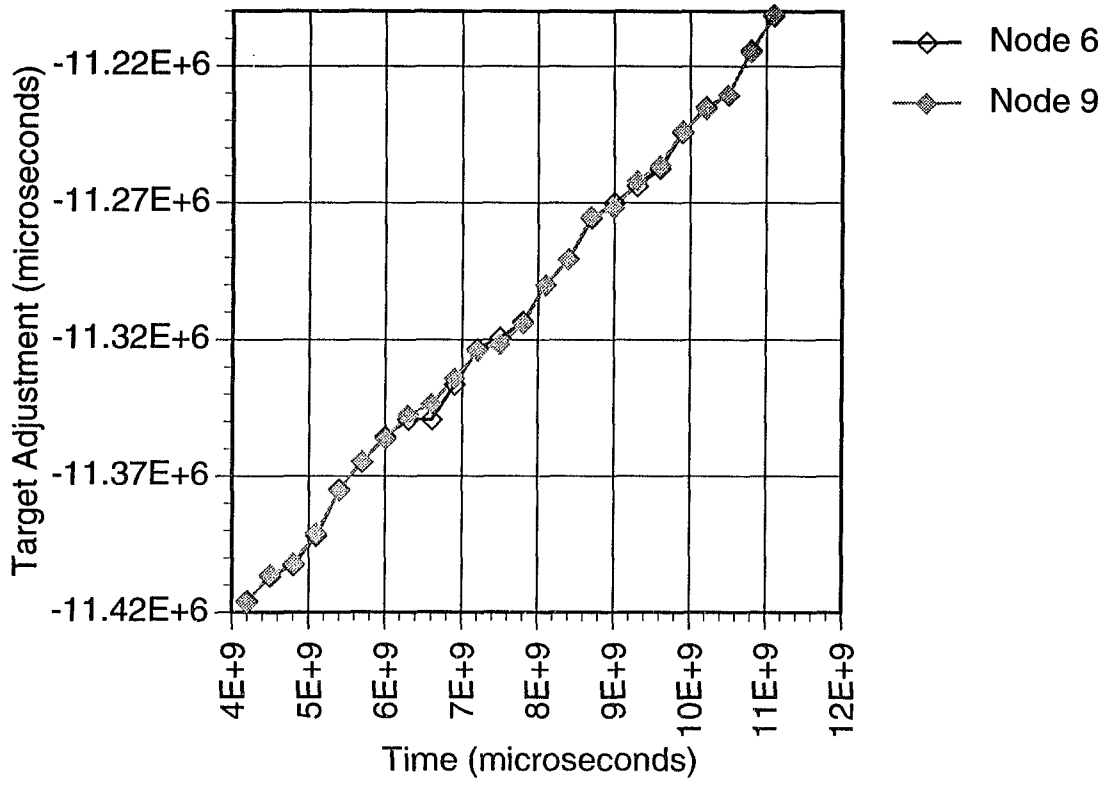

Figure 6.9: Clock adjustments of N_6 and N_9

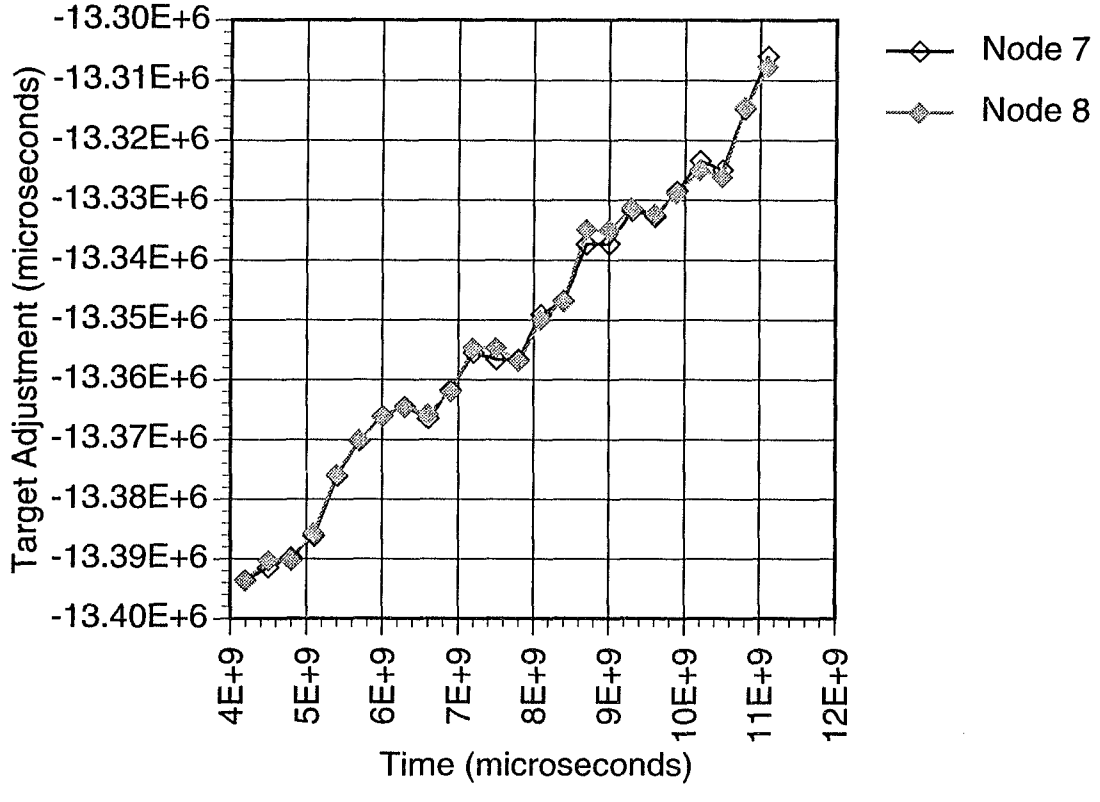

Figure 6.10: Clock adjustments of N_7 and N_8

adjustment of about 20msec. every 1000 seconds. This amounts to a drift rate of 2^{-5}, which is less than the drift rate of 5^{-5} that was used in the simulation. An increase in the minimum synchronization adjustment would slow the rate, determining the average estimate bias and subtracting it from the synchronization adjustment is another possibility.

6.6 Summary

A continuously-operating synchronization algorithm would have several advantages. It would not generate the periodic load characteristic of standard synchronization algorithms, it would allow continuous monitoring of skews, and would let nodes adjust their clocks quickly in response to a change in skew. Unfortunately, continuous operation also increases the cost of distributing clock information. This makes it difficult to use probabilistic estimation algorithms and peer adjustment algorithms with continuous operation.

This chapter introduced an efficient continuous clock distribution algorithm that may be used with the probabilistic estimation algorithms of Chapter 5. Peer adjustment algorithms may be used with the resulting estimates. The principle features of this algorithm are the following:

- Continuous distribution of clock information.
- A flexible means of defining how clock information is to be distributed that allows for a wide variety of message sending patterns.
- Works well with probabilistic estimation algorithms, allowing low-uncertainty estimates and tight synchronization.
- Can distribute clock information system-wide, allowing every node to estimate every other node, making peer synchronization possible.

CHAPTER 7

SUMMARY AND FUTURE WORK

This dissertation has considered a great many of the problems associated with synchronization of distributed systems, and has presented new solutions for many of them. However it is not the last word on synchronization. Work on synchronization is likely to continue for many years yet. This chapter summarizes the contributions of this dissertation, and lists some of the work that still remains to be done.

7.1 Summary

Synchronizing the clocks of a distributed system is a difficult problem that involves trade-offs between maximum skew and network load. There is no general synchronization algorithm which works well under all circumstances.

Part of the reason for this is that synchronization algorithms usually have three distinct parts: distribution of clock information, estimation of clock skews, and adjustment of the local clock. Each of these three parts interacts with the system differently. Breaking synchronization into these three operations, and developing separate algorithms for each part, allows some freedom to choose algorithms for each operation that are well suited to the system under consideration.

This dissertation takes the above approach, and several algorithms for each of the three parts of synchronization are presented. These include:

- A clock adjustment algorithm which takes into account variation in the quality of estimates and does not require estimates for every other node.
- A clock adjustment algorithm which allows synchronization to be maintained while requiring each node make estimates of only a subset of the other nodes.

- An efficient clock distribution algorithm which greatly reduces the number of synchronization messages needed by probabilistic estimation algorithms, and two probabilistic estimation algorithms that work with it.

- An algorithm for continuously distributing clock information, thus allowing for continual estimation of clock skews, and more frequent clock adjustments.

As stated above, these algorithms are only for parts of synchronization, none constitutes a synchronization algorithm in its own right. They can, however, be combined to make a synchronization algorithm. The adjustment algorithms of Chapters 3 and 4 can be combined with the clock distribution and estimation algorithms of Chapters 5 and 6. For example, the continuous clock distribution algorithm could be combined with the interval estimation algorithm, and used in the synchronization group environment of Chapter 4.

These algorithms need not be combined only with one another. Other algorithms may be used as well. The adjustment algorithms of Chapters 3 and 4 may be used with simple (i.e., non-probabilistic) estimation algorithms like the one in [22]. The distribution and estimation algorithms of Chapters 5 and 6 may be used with an interactive convergence adjustment algorithm, or even with a master/slave adjustment algorithm.

The flexibility of this work is such that it may be taken whole, or piecemeal, depending on the needs of the system.

7.2 Future Work

There is little work left to do on the algorithms themselves. Much of the remaining work involves further simulation and implementation. There are, however, several new but related projects in progress. Specific areas of future work are the following:

- More simulation work needs to be done, especially of the continuous synchronization algorithm. Simulation is to be done for larger systems, and different system geometries. Simulation of faults may also be added.

- The continuous synchronization simulator that runs on workstations is in the process of being ported to a VXWorks system. This is a prelude to eventual implementation as part of the computer system in the Titan IV launch system [17].

- Some consideration should be given to the use of the adjustment algorithm of Chapter 4 with the estimation algorithms of Chapters 5 and 6. Theoretically, they would

work well together, but the needs of the adjustment algorithm of Chapter 4 are somewhat different than those of Chapter 3. Specifically, the algorithm of Chapter 4 cannot discard estimates simply because their uncertainty is too high. This may mean the estimation algorithms must work harder to reduce the maximum estimation uncertainty.

- NTP uses its estimates to try to estimate the rate at which skew is increasing. Nodes can then try to speed up or slow down their clocks in order to reduce this rate. This reduces the need for synchronization since clocks do not drift apart as quickly. Implementation and analysis of such an algorithm is relatively easy with a master/slave adjustment algorithm like NTP. It is not so easy with a peer adjustment algorithm.

BIBLIOGRAPHY

[1] J. E. Abate, E. W. Butterline, R. A. Carley, P. Greendyk, A. M. Montenegro, C. D. Near, S. H. Richman, and G. P. Zampetti, "AT&T's new approach to the synchronization of telecommunication networks," *IEEE Communications Magazine*, pp. 35–45, April 1989.

[2] K. Arvind, "A new probabilistic algorithm for clock syncronization," in *Proc. Real-Time Systems Symposium*, pp. 330–339, Santa Monica, CA, 1989.

[3] K. Arvind, "A new probablistic algorithm for clock synchronization," Technical Report 89-86, University of Massachusetts, Amherst, August 1989. Available in RTCL library.

[4] R. E. Beehler and D. W. Allan, "Recent trends in NBS time and frequency distribution services," *Proceedings of the IEEE*, vol. 74, no. 1, pp. 155–157, January 1986.

[5] W. R. Braun, "Short term frequency instability effects in networks of coupled oscillators," *IEEE Trans. Communications*, vol. COM-28, no. 8, pp. 1269–1275, August 1980.

[6] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt, *First leaves: a tutorial introduction to Maple V*, Springer-Verlag, New York, 1992.

[7] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt, *Maple V library reference manual*, Springer-Verlag, New York, 1991.

[8] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt, *Maple V language reference manual*, Springer-Verlag, New York, 1991.

[9] M.-S. Chen, K. G. Shin, and D. D. Kandlur, "Addressing, routing and broadcasting in hexagonal mesh multiprocessors," *IEEE Trans. Computers*, vol. 39, no. 1, pp. 10–18, January 1990.

[10] C. C. Costain, "Time transfer via geostationary satellites," *Proceedings of the IEEE*, vol. 74, no. 1, pp. 161–162, January 1986.

[11] F. Cristian, "Probabilistic clock synchronization," *Distributed Computing*, vol. 4, no. 3, pp. 146–158, 1989.

[12] R. E. Drullinger, "Frequency standards based on optically pumped cesium," *Proceedings of the IEEE*, vol. 74, no. 1, pp. 140–142, January 1986.

[13] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum flow problem," in *Proc. 18th Annual ACM Symp. on Theory of Computing*, pp. 136–146, 1986.

[14] J. Y. Halpern, B. Simons, R. Strong, and D. Dolev, "Fault-tolerant clock synchronization," in *Proc. 3rd Symp. on Principles of Distributed Computing*, pp. 89–102, 1984.

[15] J. Y. Halpern and I. Suzuki, "Clock synchronization and the power of broadcasting," *Distributed Computing*, vol. 5, no. 2, pp. 73–82, 1991.

[16] A. L. Hopkins, Jr., T. B. Smith, III, and J. H. Lala, "FTMP-A highly reliable fault-tolerant multiprocessor for aircraft," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1221–1239, October 1978.

[17] B. J. Jambor and G. W. Eger, "Fault tolerant high speed network for space systems," in *Proc. MILCON '88*, pp. 13.4.1–13.4.8, San Diego, 1988.

[18] A. Kajackas, "On synchronization of communication networks with varying channel delays," *IEEE Trans. Communications*, vol. COM-28, no. 8, pp. 1267–1268, August 1980.

[19] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[20] J. L. W. Kessels, "Two designs of a fault-tolerant clocking system," *IEEE Trans. Computers*, vol. C-33, no. 10, pp. 912–919, October 1984.

[21] C. M. Krishna, K. G. Shin, and R. W. Butler, "Ensuring fault tolerance of phase-locked clocks," *IEEE Trans. Computers*, vol. C-34, no. 8, pp. 752–756, August 1985.

[22] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *Journal of the ACM*, vol. 32, no. 1, pp. 52–78, January 1985.

[23] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, July 1982.

[24] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[25] W. C. Lindsey and A. V. Kantak, "Network synchronization of random signals," *IEEE Trans. Communications*, vol. COM-28, no. 8, pp. 1260–1266, August 1980.

[26] J. C. Luetchford, J. Heynen, and J. W. Bowick, "Synchronization of a digital network," *IEEE Trans. Communications*, vol. COM-28, no. 8, pp. 1285–1290, August 1980.

[27] J. Lundelius and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," in *Proc. Principles of Dist. Comput.*, pp. 75–88, June 1984.

[28] M. Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems," *ACM Trans. Computer Systems*, vol. 3, no. 2, pp. 145–159, May 1985.

[29] A. J. Martin, "The torus: An exercise in constructing a processing surface," in *Proc. Caltech Conf. on VLSI*, pp. 527–537, 1981.

[30] K. Marzullo and S. Owick, "Maintaining the time in a distributed system," *Operating Systems Review*, vol. 19, no. 3, pp. 44–54, July 1985.

[31] D. L. Mills, "On the accuracy and stability of clocks synchronized by the network time protocol in the internet system," *Computer Communication Review*, vol. 20, no. 1, pp. 65–75, January 1990.

[32] D. L. Mills, "Network time protocol (Version 3) specification, implementation and analysis," Request for Comments 1305, University of Deleware, March 1992.

[33] D. Mitra, "Network synchronization: Analysis of a hybrid of master-slave and mutual synchronization," *IEEE Trans. on Communcations*, vol. COM-28, no. 8, pp. 1245–1259, August 1980.

[34] A. G. Mungall, "Frequency and time — national standards," *Proceedings of the IEEE*, vol. 74, no. 1, pp. 132–136, January 1986.

[35] P. Ramanathan, D. D. Kandlur, and K. G. Shin, "Hardware assisted software clock synchronization for homogeneous distributed systems," *IEEE Trans. Computers*, vol. C-39, no. 4, pp. 514–524, April 1990.

[36] S. Rangarajan and S. K. Tripathi, "Efficient synchronization of clocks in a distributed system," in *Proc. Real-Time Systems Symposium*, pp. 22–31, December 1991.

[37] N. W. Rickert, "Non byzantine clock synchronization — a programming experiment," *Operating Systems Review*, vol. 22, no. 1, pp. 73–78, January 1988.

[38] C. L. Seitz, "The cosmic cube," *Communications of the ACM*, vol. 28, no. 1, pp. 22–33, January 1985.

[39] K. G. Shin and P. Ramanathan, "Clock synchronization of a large multiprocessor system in the presence of malicious faults," *IEEE Trans. Computers*, vol. C-36, no. 1, pp. 2–12, January 1987.

[40] K. G. Shin and P. Ramanathan, "Transmission delays in hardware clock synchronization," *IEEE Trans. Computers*, vol. 37, no. 11, pp. 1465–1467, November 1988.

[41] T. K. Srikanth and S. Toueg, "Optimal clock synchronization," *Journal of the ACM*, vol. 34, no. 3, pp. 626–645, July 1987.

[42] N. Vasanthavada and P. N. Marinos, "Synchronization of fault-tolerant clocks in the presence of malicious failures," *IEEE Trans. Computers*, vol. 37, no. 4, pp. 440–448, April 1988.

[43] F. L. Walls, "Frequency standards based on atomic hydrogen," *Proceedings of the IEEE*, vol. 74, no. 1, pp. 142–146, January 1986.

[44] D. J. Wineland, "Frequency standards based on stored ions," *Proceedings of the IEEE*, vol. 74, no. 1, pp. 147–150, January 1986.

[45] G. M. R. Winkler, "Changes at USNO in global timekeeping," *Proceedings of the IEEE*, vol. 74, no. 1, pp. 151–155, January 1986.

[46] S. Wolfram, *Mathematica: a system for doing mathematics by computer*, Addison Wesley, Redwood City, California, second edition, 1990.

[47] K. Yoshimura, M. Imae, M. Urazuka, T. Morikawa, T. Yoshino, S. Kobayashi, and T. Igarashi, "Research activities on time and frequency transfer usin space links," *Proceedings of the IEEE*, vol. 74, no. 1, pp. 157–160, January 1986.