Marching to Many Distant Drummers

Leslie Lamport and Timothy Mann

Mon 26 May 1997 [18:27]

Abstract

We address the problem of determining the time in a network where a node may obtain information indirectly from primary time sources via intermediate nodes. Our key idea is to transmit and store each time datum as a pair, consisting of a time interval and a "failure predicate", a boolean expression that indicates precisely which combinations of node failures could invalidate the interval. We describe some techniques based on this idea, but not a complete system design or implementation.

Contents

1 Introduction

Published algorithms for fault-tolerant clock synchronization assume a fixed set of primary time sources, which are read directly by a node seeking to determine the current time [1, 3, 4]. Here, we address the problem of determining the time in a network when a node may obtain information indirectly from primary time sources via intermediate nodes.

We assume that a client of a time service is provided with an interval that is supposed to contain UT, the correct time according to some time standard. The time service has failed if it provides a client with an interval that does not contain UT.

To appreciate the problem, suppose a node A obtains two intervals I_1 and I_2 directly from two primary time sources. Node A can conclude that, in the absence of more than one failure, UT is in at least one of the two intervals. By providing a client with an interval that contains both I_1 and I_2, the node A implements a time service that can tolerate a single failure.

However, suppose that those two intervals did not come directly from the two primary time sources, but were both relayed by a single intermediate node. Then failure of that intermediate node could make both intervals incorrect. Hence, the two intervals do not provide node A with enough information to implement a time service that can tolerate a single failure.

To implement a practical distributed time service, one must make a number of choices based on the characteristics of the network. Instead of presenting particular algorithms, we develop a mathematical foundation for deriving these algorithms. Section 2 introduces a simple calculus of *time data* for keeping track of information about the correct time and of the source of that information. Section 3 describes techniques for using time data in a distributed time service. The following two sections analyze the costs of these techniques.

In the following discussion, we consider "fault" and "failure" to be synonymous.

2 A Calculus of Time Data

2.1 Node predicates

Assume a set of nodes, each with a unique name. A *node predicate* is a positive Boolean combination of node names—that is, a boolean formula without negation. In writing node predicates, we denote conjunction by $\cdot$ and disjunction by $+$, so if A, B, and C are node names, then $A \cdot (B + A \cdot C)$, which equals $A \cdot B + A \cdot C$, is a node predicate. We denote *true* by 1 and *false* by 0.

We interpret a node name as an elementary proposition asserting that the node has failed. Thus, the node predicate $A \cdot B + A \cdot C$ asserts that both A and B have failed or both A and C have failed.

A *term* is the product of node names—for example, $A \cdot B$. A node predicate is in *normal form* if it is the sum of a minimal number of terms—for example, $A \cdot B + A \cdot C$. There is a unique normal form for any node predicate. A *linear* node predicate is one whose normal form is the sum of node names—for example, $A + C + E$.

The *degree* $\deg(T)$ of a term T is the number of distinct nodes in T. We consider 1 to be a term of degree 0 and consider 0 to be a term of degree ∞. The degree $\deg(F)$ of a nonzero node predicate F is the minimum degree of the terms in its normal form. Thus, $\deg(A + B \cdot C)$ equals 1. The degree $\deg(F)$ of F is the smallest number of node failures that can make F true.

For any node predicates F and G with $G \neq 0$, we define $\deg(F|G)$, the *degree of* F *relative to* G, to equal $\deg(F \cdot G) - \deg(G)$. Thus, $\deg(F|G)$ more node failures are required to make both F and G true than are required to make G true. Observe that $\deg(F|1) = \deg(F)$.

For any node predicate F and term T, define $F[1/T]$ and $F[0/T]$ to be F with 1 or 0, respectively, substituted for all the nodes in the term T. Then $\deg(F[1/T])$ and $\deg(F[0/T])$ are the number of failures needed to make F true assuming that the nodes in T all have, or all have not, already failed.

Suppose we interpret a node predicate F as an ordinary polynomial in the node names. If, in this polynomial, we substitute for each node name A an *a priori* probability that A fails, then the result is the probability that F is true—ignoring terms of order greater than $\deg(F)$. For example, the probability that the node predicate $A \cdot C + B \cdot C$ is true is given by the polynomial $A \cdot C + B \cdot C - A \cdot B \cdot C$; it differs from $A \cdot C + B \cdot C$ by a polynomial of degree 3, which equals $\deg(A \cdot C + B \cdot C) + 1$. (This follows from the fact that if events e_1 and e_2 have probability p_1 and p_2, then e_1 or e_2 occurs with probability $p_1 + p_2 - p_1 p_2$.)

For the purpose of computing failure probabilities, we make the following assumptions:

- The *a priori* probability that a node is faulty, without its failure being detected, is at most p.
- The number of distinct node names that appear in the node predicates under consideration is at most N, where $Np \ll 1$.

Let $\|F\|$ equal $mp^{\deg(F)}$, where m is the number of distinct terms of order $\deg(F)$ in the normal form of F. Under our assumptions, $\|F\|$ is, to within a factor of $(1+Np)$, the *a priori* probability that F is true. We define $\|F|G\|$ to be $\|F \cdot G\|/\|G\|$, the approximate probability that F is true given that G is true.

Note that $p^{\deg(F)} \leq \|F\| \leq (Np)^{\deg(F)}$, so $\|F|G\| \leq (Np)^{\deg(F|G)} N^{\deg(G)}$. We will usually assume that N is so much smaller than $1/p$ that we can ignore factors of N and take $p^{\deg(F|G)}$ to be the probability that F is true given that G is true. We therefore use $\deg(F|G)$ rather than $\|F|G\|$ to measure the likelihood that F is true given that G is true. This assumption is not deeply embedded in our algorithms, and it would be easy to replace our use of degrees by the use of probabilities, but we feel that this will seldom be necessary.

To explain why we can usually ignore factors of N in this way, recall that p is not the probability that a node has failed—which is all too large with current systems—but that the node has failed without its failure being detected. Such undetected failures are much less likely than ordinary "crashes", which are easily detected, so we expect p to be quite small. Actually, it is not even necessary to detect crashes—the failure of a node to do anything—in our algorithms, since nodes that do nothing cannot lead to errors. As for N, even though there may be many thousands of nodes in the entire network, only a small number of them will take part in calculating the value provided to any individual client. The names of only those nodes that take part in the calculation will appear in the node predicates.

For greater generality, we could assume a degree $d(A)$ for each node A such that the *a priori* failure probability for each node is $p^{d(A)}$. We would then have to change our definitions of degree for $p^{\deg(F|G)}$ to remain the approximate probability of F given G. This would make the calculations more complicated. Since we do not expect useful information about actual failure probabilities to be available in a real network, we will not pursue this generalization.

We list the following simple results for later use, where F and G are any node predicates and T is any term. These results are all simple consequences of the definitions.

$$\deg(F) - \deg(G) \leq \deg(F|G) \leq \deg(F) \tag{1}$$

$$\deg(F[1/T]) \leq \deg(F) \leq \deg(F[1/T]) + \deg(T) \tag{2}$$

$$(F+G)[1/T] = F[1/T] + G[1/T] \tag{3}$$

$$(F \cdot G)[1/T] = F[1/T] \cdot G[1/T] \tag{4}$$

$$\deg(F \mid T \cdot G) = \deg(F[1/T] \mid G[1/T]) \tag{5}$$

2.2 Time data

A *time interval* I is an interval on the real line of the form $[L, R]$, with $L \leq R$. We consider a time interval I to represent the assertion $UT \in I$, that the correct time lies in the interval I. For any time intervals I_1 and I_2, we have[1]:

$$I_1 \wedge I_2 = I_1 \cap I_2 \tag{6}$$

$$I_1 \vee I_2 = I_1 \cup I_2 \tag{7}$$

$$\mathit{false} = \emptyset \tag{8}$$

In other words, UT is in both intervals iff it is in their intersection; it is in one of the intervals iff it is in their union; and the assertion that UT is in the empty interval is false.

A *time datum* is a pair (I, F), where I is a time interval and F is a node predicate. We call F the *failure predicate* of the datum. The datum represents the logical formula $I \vee F$, which asserts that UT is in I or F is true. (More precisely, it asserts that UT is in I or node failures have occurred that make F true.)

From now on, we introduce the following conventions, where j is any natural number:

- I and I_j denote time intervals.
- F and F_j denote node predicates.
- D_j denotes the time datum (I_j, F_j).

We now consider how to infer information from time data. Most of our inferences will be based on the following two tautologies, which are simple consequences of (6–8) and the definitions.

$$D_1 \wedge D_2 \Rightarrow (I_1 \cap I_2, F_1 + F_2) \tag{9}$$

$$D_1 \wedge D_2 \Rightarrow (I_1 \cup I_2, F_1 \cdot F_2) \tag{10}$$

[1]Only in node predicates do we use the boolean algebra notation $+$, $\cdot$, and 0 for $\vee$, $\wedge$, and *false*.

2.2.1 Extracting failure knowledge from time data

From (9) and (10), we see that if $I_1 \cap I_2$ is empty, then $D_1 \wedge D_2$ implies $F_1 + F_2$. In other words, if two time data make contradictory assertions about the value of UT, then one of their failure predicates is true. We generalize this observation to make the following definition of $FK(D_1, \ldots, D_n)$, which represents the knowledge about failures that can be inferred from the data D_j. (In these definitions, we ignore the possibility that some of the time intervals may be empty. Time data with empty intervals do not arise.)

$$FK(D_1) \triangleq 1 \tag{11}$$

$$FK(D_1, D_2) \triangleq \begin{cases} F_1 + F_2 & \text{if } I_1 \cap I_2 = \emptyset \\ 1 & \text{otherwise} \end{cases} \tag{12}$$

$$\text{If } n > 2,\ FK(D_1, \ldots, D_n) \triangleq \prod_{i,j} FK(D_i, D_j) \tag{13}$$

From (9) and (10) we infer

$$D_1 \wedge \ldots \wedge D_n \Rightarrow FK(D_1, \ldots, D_n)$$

2.2.2 Extracting one time datum from many

We now consider how to deduce information from a collection of time data. Given a collection $D_1, \ldots, D_n$ of time data, we want to extract a single "best" datum D that is implied by $D_1 \wedge \ldots \wedge D_n$. A time datum (I, F) is made "better" (containing more information) by making the interval I narrower and by making F stronger. Thus, there are two natural measures of quality of the time datum: the width of I, and the degree $\deg(F|FK)$, where FK is the failure knowledge. (In general, the failure knowledge FK will equal $G \cdot FK(D_1, \ldots, D_n)$, where G is knowledge that might have been obtained from other sources.) Making I narrower means specifying the value of UT more precisely. Raising $\deg(F|FK)$ makes it more likely that the value of UT specified by I is correct (by making it less likely that the number of node failures needed to make F true have occurred). Because of these two different measures of quality of a time datum, trying to combine data into a single "best" datum in general involves a compromise between precision (width of I) and reliability ($\deg(F|FK)$).

The following two rules provide useful cases in which two time data can be combined into one with no loss of information. They are simple consequences of the definitions and (6–8).

$$\text{If } F_1 \Rightarrow F_2 \text{ and } I_1 \subseteq I_2 \text{ then } D_1 \wedge D_2 = D_1 \tag{14}$$

$$\text{If } F_1 = F_2 \text{ then } D_1 \wedge D_2 = (I_1 \cap I_2, F_1) \tag{15}$$

We now give a more general method of combining the data $D_1, \ldots, D_n$ into a single datum D. The method involves separately combining the information in the left and right endpoints of the intervals I_j. For this derivation, let I_j be the interval $[L_j, R_j]$, for each j.

The left endpoint L of an interval contains the information that $UT \geq L$. Thus, a left endpoint is "better" iff it is bigger. Let b be a permutation of the integers $1, \ldots, n$ such that $L_{b(j)}$ is the jth best left endpoint among all the L_i. In other words,

$$L_{b(1)} \geq L_{b(2)} \geq \ldots \geq L_{b(n)} \tag{16}$$

Similarly, we define $e(k)$ to be the kth best right endpoint, so

$$R_{e(1)} \leq R_{e(2)} \leq \ldots \leq R_{e(n)} \tag{17}$$

(If some of these endpoints are equal, so b and e are not uniquely determined, choose any such b and e. We will see later that this choice makes no difference.)

By repeated application of (10), we find for $1 \leq j \leq n$ and $1 \leq k \leq n$:

$$D_{b(1)} \wedge \ldots \wedge D_{b(j)} \Rightarrow (I_{b(1)} \cup \ldots \cup I_{b(j)}, F_{b(1)} \cdots F_{b(j)})$$

$$D_{e(1)} \wedge \ldots \wedge D_{e(k)} \Rightarrow (I_{e(1)} \cup \ldots \cup I_{e(k)}, F_{e(1)} \cdots F_{e(k)})$$

Since $D_1 \wedge \ldots \wedge D_n$ implies both $D_{b(1)} \wedge \ldots \wedge D_{b(j)}$ and $D_{e(1)} \wedge \ldots \wedge D_{e(k)}$, we infer

$$\begin{aligned} &D_1 \wedge \ldots \wedge D_n \Rightarrow \\ &\quad (I_{b(1)} \cup \ldots \cup I_{b(j)}, F_{b(1)} \cdots F_{b(j)}) \wedge (I_{e(1)} \cup \ldots \cup I_{e(k)}, F_{e(1)} \cdots F_{e(k)}) \end{aligned}$$

Applying (9) then gives

$$D_1 \wedge \ldots \wedge D_n \Rightarrow (I, F_{b(1)} \cdots F_{b(j)} + F_{e(1)} \cdots F_{e(k)}) \tag{18}$$

where we define

$$I \triangleq (I_{b(1)} \cup \ldots \cup I_{b(j)}) \cap (I_{e(1)} \cup \ldots \cup I_{e(k)})$$

From (16) and (17), it follows that the left endpoint of $I_{b(1)} \cup \ldots \cup I_{b(j)}$ is $L_{b(j)}$ and the right endpoint of $I_{e(1)} \cup \ldots \cup I_{e(k)}$ is $R_{e(k)}$. Therefore, $I \subseteq [L_{b(j)}, R_{e(k)}]$, so (18) gives

$$D_1 \wedge \ldots \wedge D_n \Rightarrow ([L_{b(j)}, R_{e(k)}], F_{b(1)} \cdots F_{b(j)} + F_{e(1)} \cdots F_{e(k)}) \tag{19}$$

We define

$$\mathrm{MLM}_{j,k}(D_1, \ldots, D_n) \triangleq ([L_{b(j)}, R_{e(k)}], F_{b(1)} \cdots F_{b(j)} + F_{e(1)} \cdots F_{e(k)}),$$

so (19) becomes

$$D_1 \wedge \ldots \wedge D_n \Rightarrow \mathrm{MLM}_{j,k}(D_1, \ldots, D_n).$$

Increasing j and k makes the failure predicate of $\mathrm{MLM}_{j,k}(D_1, \ldots, D_n)$ stronger (making the time interval more likely to be correct), but it widens the time interval (making it contain less information about UT). Thus, choosing j and k involves a tradeoff between the precision and the reliability of the information about UT contained in the datum. To combine a collection $D_1, \ldots, D_n$ of time data into a single datum, one chooses the smallest j and k so that the failure predicates $F_{b(1)} \cdots F_{b(j)}$ and $F_{e(1)} \cdots F_{e(k)}$ have large enough degrees relative to the failure knowledge FK.

We now show that, if b and e are not uniquely determined because some of the endpoints are equal, then the choices for b and e do not matter. Suppose the jth through $(j+m)$th best left endpoints are equal. Then $\mathrm{MLM}_{j+m,k}(D_1, \ldots, D_n)$ has the same interval as $\mathrm{MLM}_{j,k}(D_1, \ldots, D_n)$ and a failure predicate that is at least as strong, so

$$\mathrm{MLM}_{j+m,k}(D_1, \ldots, D_n) \Rightarrow \mathrm{MLM}_{j,k}(D_1, \ldots, D_n)$$

In this case, there is never any reason to use $\mathrm{MLM}_{j,k}(D_1, \ldots, D_n)$ rather than $\mathrm{MLM}_{j+m,k}(D_1, \ldots, D_n)$. This implies that the choice of permutation b does not matter. A similar comment applies to right endpoints and the permutation e.

2.3 Fault-tolerance versus reliability: a digression

Fault-tolerance of an algorithm is usually measured in terms of the number of failures that it can tolerate. Suppose a time server correctly deduces a datum (I, F) and reports to a client that UT is in the interval I. The datum implies that the value reported to the client is correct unless $\deg(F)$ failures have occurred, so the algorithm would be said to tolerate $\deg(F)$ failures.

However, the purpose of fault-tolerance is to guarantee a sufficiently low probability that the service provides an incorrect value. Under our assumptions about *a priori* failure probabilities, the datum (I, F) allows us to conclude that, in the absence of other information, the probability that I is an incorrect interval is, neglecting factors of N, of order $p^{\deg(F)}$. However, additional information may be present. If we have failure knowledge FK, then the probability that I is incorrect is of order $p^{\deg(F|FK)}$, not $p^{\deg(F)}$. This is because, once we know that $\deg(FK)$ failures have occurred, the probability that k more failures will occur is of order p^k, not $p^{\deg(FK)+k}$.

Many of the fault-tolerant algorithms in the literature are predicated on the assumption that there will be at most some fixed number k of faults. If k faults are observed to have occurred, then the algorithms can act as if no further faults are possible. This assumption of at most k faults can be justified because it produces simpler algorithms, but it does not yield the most reliable algorithms. A more reliable algorithm is one that attempts to detect faults and, after a fault has been detected, can tolerate k more faults. This is the type of algorithm that we are aiming for. Thus, we have taken $\deg(F|FK)$ rather than $\deg(F)$ as the measure of reliability of the time information provided by the datum (I, F).

3 Manipulating Time Data

Time data contains information about UT. Unlike most ordinary data, information about UT tends to degrade both with the passage of time and when the information is transmitted from one place to another. We now study this degradation.

First, we introduce some notation. For any interval $I = [L, R]$ and real numbers t and w, with $w \geq 0$, define $T_{t,w}(I)$ to equal $[L + t - w, L + t + w]$. Thus, $T_{t,w}$ shifts an interval t units to the right, and widens it by $2w$ units. We extend $T_{t,w}$ to a mapping on time data by letting $T_{t,w}((I, F)) = (T_{t,w}(I), F)$.

3.1 Maintaining time data at a node

A time datum (I, F) asserts that $UT \in I$. Since UT is changing, the datum is expected to be correct only at some specific instant. We first examine what we can deduce from the knowledge that a datum was correct at an earlier instant.

We assume that each node has its own local clock. Let us suppose that the rate of a nonfaulty node's clock is known to be accurate to plus or minus

ρ seconds per second. (This is part of the definition of what makes a node nonfaulty.) If the local clock has advanced by t seconds, then UT is known to have advanced by $t \pm \rho t$ seconds. Thus, if a node knows that a time datum D is correct, then after t seconds have elapsed on its local clock, it knows only that the datum $T_{t,\rho t}(D)$ is correct. The uncertainty in its knowledge of UT has increased by $2\rho t$.

In light of this, a datum is stored at a node as a triple (I, F, c), where c is the time on the node's local clock when it knew the datum (I, F) to be correct.

3.2 Transmitting time data between nodes

Nodes exchange information by sending messages. For our purposes, it does not matter if a node requests that data be sent to it or just passively receives the data. To exchange time information, nodes must be able to determine not only the contents of a message, but also the length of time the message was in transit. Uncertainty in transmission time leads to the degradation of time information when it is sent from one node to another. To transmit a time interval with a message, a node must be able to determine bounds on the message's transmission time. If node A sends an interval I_A to B in a message whose transmission time is known by B to be $t \pm w$, then the interval I_{AB} received by B is approximately $T_{t,w}(I_A)$.

We will not consider how bounds on message transmission time are determined. We simply assume that when A sends an interval I_A to B, if $UT \in I_A$ and both A and B are nonfaulty, then B obtains an interval I_{AB} with $UT \in I_{AB}$. (We take this to be part of the definition of "nonfaulty".) Node B uses the information received from A and its knowledge of transmission delay bounds or measurement of round-trip time to compute the interval I_{AB}.

We also assume that node A can send ordinary information, such as node predicates to B, and B will receive that information correctly unless A or B is faulty. Combined with the preceding assumption, this means that if A sends a datum (I_A, F_A) to B saying that it is correct, then B knows that $(I_{AB}, A+B+F_A)$ is a correct datum. The correctness of $(I_{AB}, A+B+F_A)$ is based on the assumption that a nonfaulty node will never assert that an incorrect datum is correct. Hence, if (I_A, F_A) is incorrect, then $(I_{AB}, A + B + F_A)$ is correct because A is faulty, so $A = 1$.

There is little point to having a node participate in a time service if it believes itself to be faulty. (A node that believes itself to be faulty is faulty, since a nonfaulty node is assumed not to believe something that is not correct.) Therefore, in executing a time-service algorithm, a node assumes itself to be correct. Thus, a node B can replace any datum (I, F) by the datum $(I, F[0/B])$. In particular, when A sends the datum (I_A, F_A) to B, node B takes the datum $(I_{AB}, A + F_A[0/B])$ to be correct.

However, a node should use the information it receives from other nodes to check itself for failure. For the purposes of such checking, when B receives the datum from A, it assumes only that $(I_{AB}, A + B + F_A)$ is correct.

Time information originates at primary time sources and is transmitted between nodes. If data are not combined (except through (14) and (15)), then each retransmission of a datum adds a single node name to its failure predicate. Thus, when data are not combined, their failure predicates are linear.

3.3 Retransmitting data

Let A and B be two nodes, and suppose that their local clocks have errors in their running rates of plus or minus ρ_A and ρ_B, respectively. Suppose node A sends a datum $(I1_A, F_A)$ to node B, and t_A seconds later sends "another copy of the same" datum $(I2_A, F_A)$ to B, meaning that $I2_A$ is the "time-shifted" version $T_{t_A, \rho_A t_A}(I1_A)$ of $I1_A$.

Upon receiving the second datum, node B will have two data having the same failure predicate: the one it just received and its "time-shifted" version of the first datum. These two data can be combined using (15)—that is, B just takes the intersection of the two intervals. Node A's retransmission of "the same" data will improve B's knowledge if its time-shifted version of the first datum is not a subset of the newly received datum.

Retransmission of data in this way can be used to improve B's knowledge of UT or as a check for failures. We consider these two uses separately.

3.3.1 Refining knowledge by retransmission

Assume that A and B are nonfaulty. A simple analysis indicates that retransmission is likely to improve B's knowledge in the following cases:

1. The transmission delays were smaller the second time the datum was sent. This is most likely to improve B's datum if t_A is small. In other words, when there is significant variability in transmission delays, it can help to transmit the same datum several times in close succession.

2. The actual running rate of the two clocks differs by more than $\rho_A - \rho_B$. There are two cases of interest:

(a) $\rho_A \ll \rho_B$. In other words, if A's clock is more accurate than B's, then B can probably get better information by having A retransmit the value than by using the one it received earlier. In this case, B can maintain more precise information about UT by "storing its data on A"—sending the data to A and having A send it back from time to time. However, doing so adds A to the data's failure predicate.

(b) $\rho_A \approx \rho_B$. In this case, continual retransmission of the data will make B's copy widen at the rate of $2\rho_B - \epsilon$ rather than $2\rho_B$, where ϵ is the difference in the two clocks' rates. By "sharing their data," A and B can use the differences in their clock rates to reduce the rate at which they lose knowledge of UT. This can help only when the two clocks run at different rates, which means that at least one of them deviates from the correct rate (one second of clock time per second of UT). Sharing data in this way will not help if both clocks happen to run at exactly the same rate.

3.3.2 Detecting failures through retransmission

A failure has occurred if B's time-shifted version of the first interval it receives is disjoint from the second interval it receives. This is most likely to happen if the initial interval $I1_A$ sent by A consists of a single point. Although we can analyze this situation in terms of intervals, we instead consider the sending of local clock values. The calculations given below can be recast into the framework of node B checking for the empty intersection of two intervals $I1_{AB}$ and $I2_{AB}$ received from node A, where the first interval $I1_A$ sent by A is a single point. However, it is simpler to think in terms of local clock values.

Assume that A sends a message with its local clock value $c1_A$ to B, and B deduces that the message was sent at time $c1_B \pm w_1$ on its own local clock. Some time later, A sends another such message with its local clock value $c2_A$, which B deduces was sent at time $c2_B \pm w_2$. Neglecting higher order terms, node B can deduce that an error has occurred if

$$|(c2_B - c1_B) - (c2_A - c1_A)| > (\rho_A + \rho_B) \cdot |c2_A - c1_A| + w_1 + w_2 \qquad (20)$$

We expect gross failures that result in garbled information to be detectable by other means. We are concerned here with failures of the nodes' local clocks. We believe that two kinds of failures are most likely in a local clock: discontinuous jumps (perhaps caused by missing a clock interrupt), and running at a rate that lies outside its error bound (perhaps caused by a failure in the clock circuitry, or in the air conditioning).

From (20), it follows that discontinuous jumps are most likely to be detected if the two clock values are sent close together—that is, when $|c2_A - c1_A|$ is small—while errors in the running rate are more likely to be detected if the values are sent far apart—with a large value of $|c2_A - c1_A|$.

Error checking is performed by having A send the value c_A of its local clock every time it sends data to B. Node B can use (20) to check a newly received triple (c_A, c_B, w) against previous values that it has saved. Node B should save the previous triple (c_A, c_B, w) and the two triples having the largest values of $c_A - c_B + w$ and $c_B - c_A + w$.

If B detects a violation of (20), then it knows that either it or A has failed. In other words, it knows that $A + B$ equals 1. A decision as to whether A or B is more likely to have failed must be based on similar checks that the two nodes perform on their communication with other nodes. For example, if nodes C and D also detect a failure in their communication with A (giving failure knowledge $A + B \cdot C \cdot D$), it is unlikely that B, C, and D are all faulty.

A problematic situation arises if a violation of (20) is detected in communication between A and B, but not in the communication that they have with their neighbors. This probably means that one of the two nodes' clocks is running at a rate that is slightly outside its error bounds. It may be possible to decide which of them is faulty by seeing how close condition (20) is to being violated in the nodes' communications with their neighbors.

3.4 When to combine time data

A node in a time service will continually receive time data from other nodes. Eventually, the client must be presented with a single interval, which will be computed from a collection of time data using the function $\mathrm{MLM}_{j,k}$ for suitable j and k. Should this be done as the data arrives or upon receipt of a client request?

Storing and transmitting a time datum D transform it to $T_{t,w}(D)$ for some t and w. If a collection of data $D_1, \ldots, D_n$ are stored and transmitted together, then they are all transformed by the same function $T_{t,w}$. The relation

$$\mathrm{MLM}_{j,k}(T_{t,w}(D_1), \ldots, T_{t,w}(D_n)) = T_{t,w}(\mathrm{MLM}_{j,k}(D_1, \ldots, D_n)) \qquad (21)$$

implies that it does not matter when the function $\mathrm{MLM}_{j,k}$ is applied.

However, it does matter when we compute the failure-knowledge function FK. Although we have

$$FK(D_1, \ldots, D_n) \Rightarrow FK((T_{t,w}(D_1), \ldots, T_{t,w}(D_n))$$

the converse implication does not always hold. As the intervals widen, previously disjoint intervals can overlap, and information about what failures have occurred can be lost. Therefore, to retain the maximum amount of information, the failure knowledge $FK(D_1, \ldots, D_n)$ should be computed right away, before the intervals widen.

While this discussion might suggest that data should be combined as soon as possible, this is not the case. Although failure knowledge should be computed as soon as the data are obtained, it is not a good idea to replace a collection of data $D_1, \ldots, D_n$ by a single datum $\mathrm{MLM}_{j,k}(D_1, \ldots, D_n)$ if any additional data may become available. Such a replacement destroys potentially useful information.

Ideally, an algorithm should never combine data with a function $\mathrm{MLM}_{j,k}$ except to provide a single interval to a client. One node should send another only "raw" data, which have linear failure predicates, together with failure knowledge no longer obtainable from the data because the intervals have widened with time.

3.5 Discarding data

To maximize their knowledge, nodes would keep sending each other all their data and would never discard any old data. This obviously cannot go on forever; some data must eventually be discarded or combined with other data. We now consider how that is done.

3.5.1 Discarding stale data

Rule (15) can be applied to replace two data having the same failure predicate with a single datum. A datum's failure predicate indicates the paths traveled by the information contained in the datum. For example, if the datum $(I, X + A + B)$ is known by a node C, where X is a primary time source, then the fact that UT lies in I was derived by C from information that it received from X along a path consisting of nodes A and B.

Two data containing information that traveled along the same paths have the same failure predicate. One datum is probably a more recent version of the other. If node C has two data $(I_1, X+A+B)$ and $(I_2, X+A+B)$, then these are probably derived from data that X sent at two different times.

Perhaps $(I_2, X + A + B)$ was just received and $(I_1, X + A + B)$ is the time-shifted version of older data. In this case, since I_1 has widened with time (Section 3.1), it is likely to be a subset of I_2, so rule (15) becomes a special case of rule (14) and simply replaces the first datum by the second. Rule (15) is the means by which fresh data replaces older data.

3.5.2 Discarding data that has gone too far

Rule (14) permits one datum to be discarded in favor of another that has traveled along a shorter path. Suppose that a node C has two data $(I_1, X + A)$ and $(I_2, X + A + B)$. These are probably the same datum sent from X via two different paths—the first containing just node A and the second containing nodes A and B. (Perhaps A relayed the second datum from B before relaying the first directly from X.) Since each retransmission of a datum widens its interval, it is likely that I_1 is a subset of I_2, so (14) can be applied to discard the second datum.

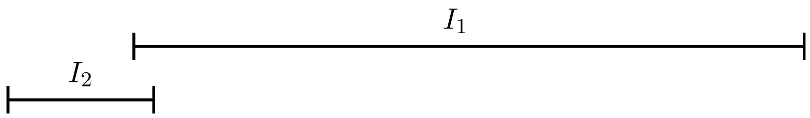
In principle, data could be sent along all possible, arbitrarily long paths. Rule (14) would prevent data from proliferating by allowing data that traveled along unnecessarily long paths to be discarded. However, in practice, a time-service algorithm would simply not send data along very long paths.

3.5.3 Discarding the worst data

Rules (14) and (15) provide the only ways to discard data without loss of information. If information must be lost, it is sensible to discard data that contain the least information. A datum (I, F) contains little information if I is wide or F is likely to equal 1. We consider these cases separately.

Discarding data with wide intervals We consider data with linear failure predicates—in particular, data that have not been combined with the $\mathrm{MLM}_{j,k}$ function. Consider a datum $(I, X + A + B)$ held by a node C, where node X is a primary time source. As we saw in Section 3.5.1, this datum is likely to be discarded by rule (15) if X keeps providing fresh data. However, suppose the datum does not get refreshed in this way, presumably because of a failure or loss of communication. Then the datum's time interval keeps getting wider. Eventually, the interval becomes so wide that it contains little information about the value of UT, and it can be discarded. More precisely, the datum should be discarded when its interval becomes very wide compared to the intervals in other data possessed by the node. Note that, since all intervals widen with time at the same rate, the interval of one datum can become relatively wider only if other intervals are "refreshed".

We will not attempt to determine a precise criterion. However, observe that a very wide interval can appear to provide useful information. In particular, this can happen if the wide interval barely overlaps another interval, as in the following example



where $I_1 \cap I_2$ is much smaller than I_2. However, if I_1 has widened with age, then it originally did not intersect I_2. In this case, we know I_1 or I_2 originally contained faulty information, so we can only hope that UT lies in either I_1 or I_2; we have no reason to expect UT to lie in their intersection.

Discarding intervals that are likely to be wrong We may want to discard data whose intervals are likely to be incorrect. The relative degree of a datum's failure predicate provides the simplest measure of how likely it is that a datum's interval is wrong. As an example, consider four time data $D_j = (I_j, F_j)$, where

$$\begin{aligned} F_1 &= A + B \\ F_2 &= C + D \\ F_3 &= C + E \\ F_4 &= E + F, \end{aligned}$$

and assume that I_1 does not intersect I_2, I_3, or I_4, but I_2, I_3, and I_4 all have a point in common. Let FK denote the failure knowledge $FK(D_1, D_2, D_3, D_4)$. A straightforward calculation gives

$$FK = A + B + C \cdot E + C \cdot F + D \cdot E$$

$$\deg(F_1|FK) = 0$$

$$\deg(F_2|FK) = 1$$

$$\deg(F_3|FK) = 1$$

$$\deg(F_4|FK) = 1$$

This indicates that it is more likely for UT to lie outside I_1 than outside any of the other intervals, making D_1 a better candidate to discard than D_2.

However, suppose that the node obtains the additional information that E has failed. Then the node's failure knowledge becomes $FK \cdot E$, and $\deg(F_j|FK \cdot E)$ equals 0 for all j. Hence, the relative degrees indicate that UT is as likely to lie in I_1 as in any of the other intervals. (This may seem wrong, since F_3 and F_4 are known to be true, while F_1 and F_2 are not. However, the probability that F_1 is true and the probability that F_2 is true are both 1/2, and $\deg(F|FK)$ gives only the order of magnitude of the probability that F is true given that FK is true.)

In general, it is dangerous to discard a datum based on current failure knowledge because later information may reveal that discarding it was a mistake. However, if time data must be discarded, there may be no better criterion available for choosing which data to discard.

3.5.4 Using $\mathbf{MLM}_{j,k}$ to discard information

It seems reasonable to try to reduce the amount of information sent between nodes by replacing a collection of data $D_1, \ldots, D_n$ with the single datum $\mathrm{MLM}_{j,k}(D_1, \ldots, D_n)$, for the "best" choice of j and k. However, this turns out not to save much space. The problem is that the amount of space needed to hold the failure predicate of $\mathrm{MLM}_{j,k}(D_1, \ldots, D_n)$ can be of the same order of magnitude as the space needed to hold the subset of data from which it is derived—a subset consisting of up to $j + k$ elements. Thus, simply applying the function $\mathrm{MLM}_{j,k}$ is not good enough. To reduce the amount of storage required, we must simplify the failure predicate of $\mathrm{MLM}_{j,k}(D_1, \ldots, D_n)$.

The space required to hold data is analyzed in Section 4.1 below. The space required to hold a set of linear node predicates depends on the number of predicates in the set, the number of terms in each predicate (which equals the length of the path along which the information has already traveled), and the total number of distinct nodes in the set of predicates. We indicate how the number of terms in the predicates and the total number of distinct nodes can be reduced.

Assume that all nodes are partitioned into classes, with all primary time sources being in class 1. We adopt the rule that information is sent only between nodes of the same class or from a node of class i to a node of class $i + 1$. A class i node A relays to a class $i + 1$ node only a single datum, consisting of an interval derived with an $\mathrm{MLM}_{j,k}$ function from the data it has received, with a failure predicate equal to A—just as if A were the primary source of the interval.

Suppose that, under our assumptions of *a priori* failure probabilities, we want an algorithm to have failure probability of order at most p^d. (We are again neglecting factors of N.) To send data to class $i+1$ nodes, a class i node A uses an $\mathrm{MLM}_{j,k}$ function to obtain a datum (I, F) with $\deg(F|FK) \geq d$, where FK is node A's failure knowledge. Node A then sends the datum to class $i+1$ nodes as if it were a primary time source with knowledge that UT is in I.

If M classes are used and data travels along paths of about the same maximum length within each class, then this scheme reduces both the number of distinct nodes and the length of failure predicates by a factor of $1/M$. It achieves a relative failure probability of order at most p^d based on the nodes' failure knowledge. However, this is deceptive, since information that has been discarded might have provided additional failure knowledge that would have lowered the relative probability.

Another feature of this method is that a node never receives data derived from data that it transmitted—except for data explicitly identified as such by the failure predicate.

As a further refinement of this method, we propose that nodes do not combine data from different primary time sources. Our intuition suggests that the original source of the information about UT has particular importance, and should not be discarded. A class $i+1$ node combines all the data it receives from class i nodes that originated from the same primary time source X, and then acts as if it had received the resulting datum directly from X. Thus a class $i+1$ node receives from a class i node A data with failure predicates $X + A$, for primary time servers A.

This technique for reducing the amount of data transmitted seems intuitively reasonable. It is the best method we have been able to come up with for combining data before it reaches its final destination. However, we have no proof that it is in any sense optimal.

3.6 The duration of node names

In a node predicate, a node name A represents the proposition that A is faulty. But, nodes fail and are repaired, so the value of A can change over time from 0 to 1 and back to 0. In principle, each occurrence of A in a node predicate should be subscripted with a timestamp. When A sends a datum (I_A, F_A) to a node B, the datum received by B should be $(I_{AB}, A_t + F_A)$, where t denotes the time at which A sent the datum.

Timestamping node names in this way would create serious problems in computing with node predicates. We can simplify $A + A$ to A, but what about $A_t + A_{t'}$? Intuitively, we should be able to combine these terms if $|t - t'|$ is small—if A was faulty several seconds earlier it's likely still to be faulty. However, there is little reason to believe that A_t and $A_{t'}$ are equal if $|t - t'|$ is large—the fact that A was faulty several years ago tells us little about whether it is faulty now.

We eschew timestamping and adopt the approach that a node changes its name when it has reason to believe that its failure status has changed. Of course, name changing will be implemented by letting a node name consist of a node identifier together with an "epoch number," but we ignore this detail for now.

A node that detects its own failure is not going to continue participating in a time-service algorithm, so it does not need a new name. A node will assign itself a new name when it decides that it has failed and been repaired.

What should happen to data whose failure predicate contains an "obsolete" node name A? If a node name changes only after a failure of the node has been detected, then all actions taken by the node under its old name A are suspect. This suggests that 1 be substituted for A in all data. Observe that substituting 1 for A in a datum of the form $(I, A + F)$ produces the datum $(I, 1)$, which is useless and can be discarded, since it contains no information about the value of UT. Of course, such substitution can be done as soon as the node reports that it has failed, without waiting for it to choose a new name.

While it seems safe to substitute 1 for A upon learning that the node named A has failed, this might result in discarding valid information obtained from the node before it failed. For example, suppose that the node named A obtains a time datum from a primary time source X and relays that datum to node B. Node B then has a datum $(I, A + X)$. Suppose that A loses contact with X, and that this leaves B with no current source of data from X. The datum $(I, A + X)$ may contain information that is valuable to B—information that should not be discarded carelessly.

If a node B contains a datum whose failure predicate contains the name A, and B learns that the node named A has failed, it should try to decide whether the failure occurred before or after A "handled" the data. Nodes that were in direct communication with node A are in the best position to decide when A failed. Therefore, we propose that nodes that receive data directly from A be responsible for deciding if the data remain valid despite the subsequent discovery that A has failed. If node B decides that data it had received from A are invalid because of A's failure, then data it sent to another node C based on the invalid data from A are also invalid. Node B should therefore notify node C that this data is invalid. This could lead node C in turn to notify other nodes of invalid data it had sent, and so on. We will not consider the precise mechanism for such notification.

A node B must determine whether data it had received from a node A should be invalidated because of a subsequent failure of A. We propose that the decision be based upon B's monitoring of A's local clock, described in Section 3.3.2 above. If node B detects a sudden jump of A's clock, it might deduce that data received before that jump are valid. However, if node B detects a gradual drift of A's clock, indicating an incorrect running rate, then data received from A should be considered invalid if they were sent after the rate of A's clock changed. It may be quite difficult for B to determine just when A's clock rate exceeded its error bounds.

To attempt to recover information in the case of a clock running at an incorrect rate, node B could keep versions of the same datum sent by A at different times. (When node A sends a datum to node B, it can indicate whether this is a new datum or a retransmission of an old one.) Node B could keep a history of local clock values received from A to try to determine when A's clock began running at the wrong rate. However, the benefit of doing all this is probably too small to justify the complexity.

3.7 Byzantine failures

We have thus far made no assumptions about the types of failures that can occur. As we observed in Section 3.2, all the data possessed by a nonfaulty node is correct, even in the presence of "Byzantine" failures, where a faulty node can send arbitrary, malicious information to other nodes.

We believe that a robust time service should tolerate Byzantine failures in the maintenance and transmission of information about time. The proper management of temporal information requires correct real-time behavior, which is more difficult to achieve than correct functional behavior. Simple failures can result in "Byzantine behavior". For example, a node that incorrectly pauses in the middle of broadcasting a time datum can transmit conflicting data to two different sets of nodes. Thus, Byzantine failures in handling information about time should not be uncommon.

However, Byzantine failures in transmitting nontemporal information, such as node predicates, should be much less common. If suitable redundancy is used, the most likely source of mistransmission of a failure predicate is a programming error. We assume that this aspect of a node's program has been checked carefully enough to eliminate such errors. (This assumption implies a faith in programmers that we hope is warranted.) We will therefore assume that no node incorrectly transmits nontemporal data. But, before making this assumption, we consider the behavior of our algorithms in the presence of Byzantine failures in the transmission of node predicates.

In Section 3.2, we observed that a Byzantine failure in one node cannot cause a nonfaulty node to believe an incorrect time datum. Any time datum that node B receives from node A has a failure predicate of the form $A+F$, so the datum is correct if A is faulty. However, even though the time data received from node A are correct, if that data has incorrect failure predicates, then failure knowledge derived from the data may be incorrect. We now show that an MLM-based algorithm that ignores failure knowledge actually tolerates Byzantine failures. Ignoring failure knowledge means basing fault tolerance entirely on *a priori* failure probability, rather than on probability conditioned upon knowledge of what failures are known to have occurred. (See Section 2.3 for a discussion of why we prefer algorithms that do use failure knowledge.) Formalizing this result requires a precise definition of the concept "an MLM-based algorithm that ignores failure knowledge". We start by defining executions and algorithms.

An *execution* of an algorithm consists of a finite set $\mathcal{E}$ of events and a partial order $\rightarrow$ on $\mathcal{E}$ representing temporal precedence [2]. We assume that $\mathcal{E}$ is partitioned into sets $\mathcal{E}_{rcv}$ of *receive* events and $\mathcal{E}_{snd}$ of *send* events, and that there is a function σ from $\mathcal{E}_{rcv}$ to $\mathcal{E}_{snd}$ such that $\sigma(e) \rightarrow e$ for all $e \in \mathcal{E}_{rcv}$, where $\sigma(e)$ is the event that sends the message received at event e. We assume that an event e consists of a node e_{node} where the event occurs, a message e_{msg}, and a local state e_{st}. (The local state may include the value of the node's clock when that event occurs.) We do not require that $e_{msg} = \sigma(e)_{msg}$. Think of an event e for which $e_{msg} \neq \sigma(e)_{msg}$ as the receipt of a message that was changed in transit.

An *algorithm* is a set of executions, representing all possible executions in which all the nodes obey the algorithm.

We now define what it means for an algorithm $\mathcal{A}$ to be an MLM-based algorithm that ignores failure knowledge. The key observation is that the failure predicate of $\mathrm{MLM}_{j,k}(D_1, \ldots, D_n)$ is a sum of products of failure predicates of the D_i. We therefore define an MLM-based algorithm to be one in which the failure predicate of any sent message is the sum of products of failure predicates of messages already received, with the latter predicates modified as described in Section 3.2.

Formally, an algorithm $\mathcal{A}$ is *MLM-based* iff for every execution $\mathcal{E}$ of $\mathcal{A}$, (1) the message e_{msg} of an event e consists of a time interval e_I and a failure predicate e_F, and (2) there exists a mapping $\eta_{\mathcal{E}}$ from events in $\mathcal{E}_{snd}$ to sets of sets of events in $\mathcal{E}_{rcv}$ such that for every e in $\mathcal{E}_{snd}$:

- If f is an element of an element of $\eta_{\mathcal{E}}(e)$, then $f_{node} = e_{node}$ and $f \rightarrow e$.

- $e_F = \sum_{S \in \eta_{\mathcal{E}}(e)} \prod_{f \in S} (\sigma(f)_{node} + f_F)[0/e_{node}]$
  (We take this expression to equal 0 if $\eta_{\mathcal{E}}(e)$ is the empty set.)

For any execution $\mathcal{E}$ of an MLM-based algorithm $\mathcal{A}$, any $s \in \mathcal{E}_{rcv}$, and any failure predicate F, we define $\Delta(\mathcal{E}, s, F)$ to be the execution obtained by changing the failure predicate received by s to F, and changing all the messages sent by the receiver to reflect this change. (No other received messages are changed.) Formally, $\Delta(\mathcal{E}, s, F)$ is the execution such that:

- The events of $\Delta(\mathcal{E}, s, F)$ consist of all events e^{Δ} with $e \in \mathcal{E}$, where:
  - s^{Δ} is the same as s except with $s^{\Delta}_F = F$.
  - If $e \in \mathcal{E}_{rcv}$ and $e \neq s$, then $e^{\Delta} = e$.
  - If $e \in \mathcal{E}_{snd}$, then e^{Δ} is the same as e except

$$e^{\Delta}_F = \sum_{S \in \eta_{\mathcal{E}}(e)} \prod_{f \in S} (\sigma(f)_{node} + f^{\Delta}_F)[0/e_{node}]$$

- $\Delta(\mathcal{E}, s, F)_{rcv}$ and $\Delta(\mathcal{E}, s, F)_{snd}$ are the sets of all e^{Δ} with e in $\mathcal{E}_{rcv}$ and $\mathcal{E}_{snd}$, respectively.
- $e^{\Delta} \to f^{\Delta}$ iff $e \to f$, for all $e, f \in \mathcal{E}$.

An MLM-based algorithm $\mathcal{A}$ *ignores failure knowledge* iff $\Delta(\mathcal{E}, s, F) \in \mathcal{A}$, for every execution $\mathcal{E} \in \mathcal{A}$, send event $e \in \mathcal{E}_{rcv}$, and failure predicate F. Thus, an algorithm ignores failure knowledge iff transforming any of its executions by modifying the failure predicate of a received input value and then changing only the failure predicates sent by the receiving node yields a possible execution of the algorithm.

We model the Byzantine failure of a node by allowing the messages sent by the node to be changed arbitrarily in transit. (It doesn't matter whether the faulty behavior occurs in the node or in the wires leading from the node.) We say that an execution $\mathcal{E}$ has at most k Byzantine failures if there exists a set N containing k nodes such that $e_F = \sigma(e)_F$ for all $e \in \mathcal{E}_{rcv}$ with $\sigma(e)_{node} \notin N$. The nodes in N are considered faulty, the rest nonfaulty.

Theorem *Let $\mathcal{A}$ be an algorithm and let $\mathcal{A}'$ be the subset of $\mathcal{A}$ consisting of all executions with no Byzantine failure. If $\mathcal{A}$ is an MLM-based algorithm that ignores failure knowledge and $\mathcal{A}'$ is a correct algorithm (meaning that $P(UT \notin f_I) \leq (Np)^{\deg(f_F)}$ for each time datum (f_I, f_F) sent in $\mathcal{A}'$), then $P(UT \notin e_I) \leq 2(Np)^{\deg(e_F)}$ for the time datum (e_I, e_F) in any message sent by a nonfaulty node during an execution of $\mathcal{A}$.*

For simplicity, we prove this result in the case when there can be at most a single Byzantine failure. The generalization to multiple Byzantine failures is straightforward. We treat *a priori* failure probabilities in our usual manner (Section 2.1).

ASSUME: 1. $\mathcal{A}$ is an MLM-based algorithm that ignores failure knowledge.
2. $P(UT \notin f_I) \leq (Np)^{\deg(f_F)}$ for all events f in $\mathcal{A}'$.
3. There is at most one Byzantine-faulty node.
4. $e \in \mathcal{E}_{snd}$ and e_{msg} is nonfaulty.

PROVE: $P(UT \notin e_I) \leq 2(Np)^{\deg(e_F)}$

PROOF SKETCH: We define a sequence $\mathcal{E} = \mathcal{E}^0, \mathcal{E}^1, \ldots, \mathcal{E}^\setminus = \mathcal{E}'$ from the actual execution $\mathcal{E}$ to an execution $\mathcal{E}'$ with no Byzantine failures by correcting the transmission errors one at a time. We then prove our goal by relating $P(UT \notin e_I)$ to the probability $P(UT \notin e'_I)$ for the corresponding event e' in $\mathcal{E}'$.

⟨1⟩1. Define the natural number n and the executions $\mathcal{E}^i$ and sets S^i, for $0 \leq i \leq n$, inductively as follows.

- $\mathcal{E}^0 \triangleq \mathcal{E}$
- $S^i \triangleq \{s \in \mathcal{E}^i_{rcv} : \sigma(s)_F \neq s_F\}$
- If $S^i = \emptyset$ then $n \triangleq i$, else let s be any element of S^i minimal under $\rightarrow$ (that is, there exists no $t \in S^i$ such that $t \rightarrow s$) and let $\mathcal{E}^{i+1} \triangleq \Delta(\mathcal{E}^i, s, \sigma(s)_F)$.

For each $f \in \mathcal{E}$, we define $f^i \in \mathcal{E}^i$ inductively by letting f^{i+1} equal $(f^i)^\Delta$.

PROOF: We must prove that the induction terminates and chooses an n. Because s is chosen to be a minimal element of S^i, and $\mathcal{E}^{i+1}$ differs from $\mathcal{E}^i$ only in an event t with $s^i \rightarrow t$, it follows that $s \notin S^j$ for all $j > i$. Hence, at each step in the construction, an element is removed from S^i that does not appear in S^j for any $j > i$. Since the set of events is finite, the induction terminates.

⟨1⟩2. Let $\mathcal{E}' \triangleq \mathcal{E}^n$. Then $\mathcal{E}' \in \mathcal{A}'$

PROOF: By assumption ⟨0⟩:1 (the hypothesis assumption that $\mathcal{A}$ is an MLM-based algorithm that ignores failure knowledge), a simple induction shows that each $\mathcal{E}^i$ is in $\mathcal{A}$. By definition of S^i and because $S^n = \emptyset$, $\mathcal{E}'$ has no Byzantine failure, so it is in $\mathcal{A}'$ (by definition of $\mathcal{A}'$).

⟨1⟩3. For each $f \in \mathcal{E}$, if $0 \leq i \leq n$, then $f_{node} = f^i_{node}$, $f_{st} = f^i_{st}$, and $f_I = f^i_I$. We define f' to equal f^n, so $f' \in \mathcal{E}'$ for each $f \in \mathcal{E}$.

PROOF: Immediate from the definitions of f^i (step ⟨1⟩1) and Δ.

⟨1⟩4. For any node predicates F and G and any node A, if $F[1/A] = G[1/A]$, then $\deg(G) \geq \deg(F) - 1$.

PROOF: Since $F[1/A] = G[1/A]$ and $\deg(A)$ equals 1, equation (2) implies that both $\deg(F)$ and $\deg(G)$ lie between $\deg(F[1/A])$ and $\deg(F[1/A]) + 1$.

$\langle 1 \rangle 5$. If A is a Byzantine-faulty node, then $e_F[1/A] = e^i_F[1/A]$ for $0 \leq i \leq n$.

ASSUME: A is a Byzantine-faulty node and $0 \leq i \leq n$.

PROVE: $e_F[1/A] = e^i_F[1/A]$

$\langle 2 \rangle 1$. $e_F[1/A] = e^0_F[1/A]$

PROOF: By definition (step $\langle 1 \rangle 1$), $e^0 = e$.

$\langle 2 \rangle 2$. ASSUME: $i > 0$ and $e_F[1/A] = e^j_F[1/A]$, for all j with $0 \leq j < i$.

PROVE: $e_F[1/A] = e^i_F[1/A]$

$\langle 3 \rangle 1$. $F[1/A][0/e_{node}] = F[0/e_{node}][1/A]$, for any node predicate F.

PROOF: A is faulty (assumption $\langle 1 \rangle$) and e_{node} is nonfaulty (assumption $\langle 0 \rangle$:4), so $A \neq e_{node}$.

$\langle 3 \rangle 2$. CASE: $e^i = e^{i-1}$

PROOF: $e^i_F[1/A] = e^{i-1}_F[1/A]$, which equals $e_F[1/A]$ by the induction hypothesis $\langle 2 \rangle 2$.

$\langle 3 \rangle 3$. CASE: $e^i \neq e^{i-1}$

$\langle 4 \rangle 1$. Let f^i be the unique element of $\mathcal{E}^i_{rcv}$ that is different from the corresponding element f^{i-1} in $\mathcal{E}^{i-1}_{rcv}$. Then f^i is in $\eta_{\mathcal{E}}(e^i)$.

PROOF: f^i exists by definition of f^i (step $\langle 1 \rangle 1$) and of Δ. It is in $\eta_{\mathcal{E}}(e^i)$ by case assumption $\langle 3 \rangle$ and the definition of e^i.

$\langle 4 \rangle 2$. CASE: $\sigma(f^i)_{node} = A$

PROOF: Since e^i_F is defined to equal $(e^{i-1})^{\Delta}$, the definition of Δ implies that f^i appears only in subexpressions of the form $(\sigma(f^i)_{node} + f^i_F)[0/e_{node}]$, which by the case assumption equals $(A + f^i_F)[0/e_{node}]$. Since $(A + f^i_F)[1/A] = 1$, $\langle 3 \rangle 1$ and equations (3) and (4) imply $e^i_F[1/A] = e^{i-1}_F[1/A]$, which equals $e_F[1/A]$ by the induction assumption $\langle 2 \rangle$.

$\langle 4 \rangle 3$. CASE: $\sigma(f^i)_{node} \neq A$

$\langle 5 \rangle 1$. $\sigma(f^i)_{node}$ is nonfaulty

PROOF: Case assumption $\langle 4 \rangle$, assumption $\langle 1 \rangle$ (A is faulty), and assumption $\langle 0 \rangle$:3 (there is only a single faulty node).

$\langle 5 \rangle 2$. $f^0_F[1/A] = \sigma(f^0)_F[1/A]$

PROOF: $\sigma(f^0)_{node}$ equals $\sigma(f^i)_{node}$, so it is nonfaulty by $\langle 5 \rangle 1$. Hence, in the original execution $\mathcal{E}^0$, the predicate f^0_F is not changed in transit, so $f^0_F = \sigma(f^0)_F$.

$\langle 5 \rangle 3$. Choose j such that $0 \leq j < i$ and $\sigma(f^j)_F = \sigma(f^{i-1})_F$.

PROOF: Tim: I eliminated all parts of the statement except for what you seemed to be using. The result is nonsense, since this is trivially satisfied by letting $j = i - 1$. So, I have no idea what's problem here going on here.

$\langle 5\rangle 4.$ $\sigma(f^0)_F[1/A] = \sigma(f^j)_F[1/A]$

problem here

PROOF: Induction hypothesis $\langle 2\rangle$. Tim: This made no sense to me, since the induction hypothesis talks about e, so I don't see how to derive any conclusion about f.

$\langle 5\rangle 5.$ $\sigma(f^j)_F[1/A] = \sigma(f^{i-1})_F[1/A]$

PROOF: Step $\langle 5\rangle 3$.

$\langle 5\rangle 6.$ $\sigma(f^{i-1})_F[1/A] = f^i_F[1/A]$

PROOF: $\sigma(f^{i-1})_F = f^i_F$ by definition of Δ.

$\langle 5\rangle 7.$ $f^0_F[1/A] = f^i_F[1/A]$

PROOF: Steps $\langle 5\rangle 2$, $\langle 5\rangle 4$, $\langle 5\rangle 5$, and $\langle 5\rangle 6$.

$\langle 5\rangle 8.$ Q.E.D.

PROOF: By the definition of e^i_F (part of the definition of Δ), equations (3) and (4), and steps $\langle 3\rangle 1$ and $\langle 5\rangle 7$, we have that $e^i_F[1/A] = e^{i-1}_F[1/A]$, which equals $e_F[1/A]$ by the induction hypothesis $\langle 2\rangle$.

$\langle 4\rangle 4.$ Q.E.D.

PROOF: Immediate from $\langle 4\rangle 2$ and $\langle 4\rangle 3$.

$\langle 3\rangle 4.$ Q.E.D.

PROOF: Immediate from $\langle 3\rangle 2$ and $\langle 3\rangle 3$.

$\langle 2\rangle 3.$ Q.E.D.

This is as far as I got.

PROOF: By mathematical induction from $\langle 2\rangle 1$ and $\langle 2\rangle 2$.

$\langle 1\rangle 6.$ $\deg(e'_F) \geq \deg(e_F) - 1$

PROOF: By steps $\langle 1\rangle 4$ and $\langle 1\rangle 5$, recalling that $e'_F = e^n_F$.

$\langle 1\rangle 7.$ Q.E.D.

$\langle 2\rangle 1.$ $P(UT \notin e_I|\text{no Byzantine failures}) \leq (Np)^{\deg(e_F)}$

PROOF: If there are no Byzantine failures, then $\mathcal{E} = \mathcal{E}'$, so $(e_I, e_F) = (e'_I, e'_F)$ is a correct datum.

$\langle 2\rangle 2.$ $P(UT \notin e_I|\text{node } A \text{ has a Byzantine failure}) \leq (Np)^{\deg(e_F)-1}$

PROOF: If some node A has a Byzantine failure, then (e'_I, e'_F), the datum corresponding to e that would have been sent in the absence of the failure, is correct. Combining this observation with step $\langle 1\rangle 3$ and step $\langle 1\rangle 6$, $P(UT \notin e_I|\text{node } A \text{ has a Byzantine failure}) \leq (Np)^{\deg(e'_F)} \leq (Np)^{\deg(e_F)-1}$

$\langle 2\rangle 3.$ Q.E.D.

PROOF: Case $\langle 2\rangle 2$ occurs with *a priori* probability Np. Hence the total probability $P(UT \notin e_I) \leq (1 - Np)(Np)^{\deg(e_F)} + (Np)(Np)^{\deg(e_F)-1} \leq 2(Np)^{\deg(e_F)}$.

This result is rather weak in that it excludes algorithms that make decisions based on partial calculations. However, it can be extended to algorithms that involve decisions, so long as those decisions are based only on *a priori* probabilities. For example, consider the algorithm proposed in Section 3.5.4, in which the datum relayed from a class i node to a class $i+1$ node is obtained by applying a function $\mathrm{MLM}_{j,k}$, where j and k are chosen so the datum's failure predicate F satisfies $\deg(F|FK) \geq d$. The correctness of the algorithm rests on the fact that the probability of the resulting interval being incorrect is approximately p^d (neglecting factors of N). Now, suppose that the method is modified to choose j and k so that $\deg(F) \geq d$, ignoring the failure knowledge FK. Then the computed interval still has *a priori* probability p^d of being incorrect. The result we just proved shows that this is still the *a priori* probability in the presence of Byzantine failures. (We are neglecting the factor of 2 as well as factors of N.)

This sort of argument breaks down for an algorithm that is based on the degree $\deg(F|FK)$ of F relative to failure knowledge FK. Recall that our result rested on the observation that if F is the corrupted version of F' produced by a Byzantine failure, then $\deg(F') \geq \deg(F) - 1$. However, letting FK be the corrupted version of the failure knowledge FK', we can conclude only that $\deg(F'|FK') \geq \deg(F|FK) - 2$. This implies that Byzantine failures can corrupt failure knowledge in such a way as to invalidate correctness claims based on failure probabilities relative to correct failure knowledge. It suggests that, in the face of Byzantine failures, one cannot take advantage of failure knowledge to achieve reliability beyond that based on *a priori* failure probabilities.

From now on, we ignore the possibility of Byzantine failures in the transmission of nontemporal information. We assume that although faulty nodes can arbitrarily alter time intervals, they correctly transmit nontemporal information such as node predicates.

4 Complexity

We now consider the costs in space (message length) and time (computational complexity) of manipulating time data. We assume that the failure predicates of the time data are linear. Thus, we assume that if data are combined using the functions $\mathrm{MLM}_{j,k}$, then the resulting failure predicate is also simplified as explained in Section 3.5.4.

Rather than writing general expressions for the costs in terms of a multitude of parameters, we assume fixed values for many of the parameters. It is easy to repeat the derivations using different parameters. However, we believe that substituting other realistic choices of these parameters would not cause significant changes to the results.

4.1 Space: representing the information

The cost of sending a set of data items from one node to another depends on the number of bytes occupied by a collection of time data, which we now calculate.

4.1.1 Representing the intervals

We assume that a time is a 64-bit quantity, representing UT with a granularity of 100 nanoseconds, which equals 10^{-7} seconds. Thus,

$$\begin{aligned} 1 \text{ sec} &\approx 2^{24} \\ 1 \text{ min} &\approx 2^{30} \\ 1 \text{ hour} &\approx 2^{35} \end{aligned}$$

An interval can be written as $[s-e, s+e]$. The value of s can be any representable time. However, the width of an interval is unlikely to be known to very great precision, since it is based on the maximum error in clock rates and the measured transit delays of messages. Moreover, widening an interval by a fraction of a percent is of little consequence. Thus, it suffices to maintain e with a precision of one part in 2^{10}. A number from 0 to 2^{64} with a precision of one part in 2^{10} requires 2 bytes (using a floating-point style of representation).

Representing a single interval as $[s-e, s+e]$ requires 10 bytes—8 for s and 2 for e. However, the intervals of a set of time data will have values of s close to one another. Values of s should not differ by more than an hour. (We should be able to assume that a nonfaulty node knows the correct time to within an hour, and can therefore discard time data with intervals whose centers lie too far from what it believes UT to be.) Hence, a set of time intervals can be represented by a single 64-bit time and 36-bit offsets for each s. Thus, the time intervals in a set of $\mathcal{D}$ time data can be represented by $8 + 6.5\mathcal{D}$ bytes.

The 64-bit time used in this representation can be the value of the local clock of the node that is sending the data. Section 3.3.2 discusses the use of such local clock values in detecting failures.

4.1.2 Representing the failure predicates

We are assuming that the failure predicates are linear, so they can be represented as a list of nodes. Let $\mathcal{H}$ be the maximum length of a failure predicate (so it equals the length of the longest path along which data has traveled); let $\mathcal{D}$ be the total number of time data to be sent; and let $\mathcal{N}$ be the total number of distinct nodes in the data's failure predicates. ($\mathcal{N}$ is at most $\mathcal{D}^{\mathcal{H}}$, but we expect that it will usually be much smaller.)

If B is the number of bits needed to represent a node name, then the failure predicates can be represented with

$$\mathcal{N}B + \mathcal{H}\mathcal{D}\log\mathcal{N}$$

bits. (Each failure predicate is a list of at most $\mathcal{H}$ nodes, each of which can be identified from among the $\mathcal{N}$ nodes by $\log\mathcal{N}$ bits.) We assume that $\log\mathcal{N} \leq 8$, so $\mathcal{H}\mathcal{D}\log\mathcal{N}$ is at most $\mathcal{H}\mathcal{D}$ bytes.

We now calculate B, the number of bits needed to represent the name of a node. Assume a 48-bit node identifier, so the name of a node contains 6 bytes of information. As observed in Section 3.6, a node name will consist of the node identifier together with an epoch number, which is incremented cyclically whenever the node fails and is restarted. The number E of epoch numbers must be large enough so that data will be dropped from the system (because intervals get wide) before a node will be restarted E times. It seems adequate to let E be 2^8, so a node name plus epoch number can be represented by 7 bytes.

Putting these figures together, we find that the failure predicates can be represented by $7\mathcal{N} + \mathcal{H}\mathcal{D}$ bytes. Including the representations of the time intervals computed above, we see that transmitting a collection of time data requires at most

$$8 + (6.5 + \mathcal{H})\mathcal{D} + 7\mathcal{N} \tag{22}$$

bytes.

4.1.3 Representing the failure knowledge

Failure knowledge ultimately reduces to knowledge of pairs of intervals with disjoint intersection. We assume that the data are being passed along with the failure knowledge derived from them. (Actually, there is no need to transmit failure knowledge that can be derived from the data themselves—we need only transmit knowledge that has been lost through widening of the intervals.)

The most likely situation is for some set of data, presumably obtained through the same faulty node, all to have their intervals disjoint from the intervals of the remaining data. A reasonably compact representation of the failure knowledge should therefore be sets of pairs of subsets of the data, where the pair (S_1, S_2) indicates that the interval belonging to any datum in S_1 is disjoint from the interval belonging to any datum in S_2. Such a pair can be represented by $2\mathcal{D}$ bits, or $\mathcal{D}/4$ bytes. We expect that there will be only a small number of such pairs of sets, so the number of bytes needed to convey the failure knowledge should be a very small multiple of $\mathcal{D}$—probably less than $2\mathcal{D}$ bytes.

4.2 Computation costs

Given data $D_1, \ldots, D_n$, where D_j equals (I_j, F_j), let $\mathrm{MLM}_{j,k}(D_1, \ldots, D_n)$ equal $(I_{j,k}, F_{j,k})$, and let FK denote $FK(D_1, \ldots, D_n)$. We consider the cost of computing $I_{j,k}$ and $\deg(F_{j,k}|FK)$ for increasing j and k.

The algorithms start by sorting the intervals I_j by both their left and right endpoints, which takes $O(\mathcal{D} \log \mathcal{D})$ time. Since intervals $[L, R]$ and $[L', R']$ are disjoint iff $R < L'$ or $R' < L$, all disjoint pairs of intervals among the I_j can be found in $O(\mathcal{D} \log \mathcal{D})$ steps using the sorted lists of endpoints.

After sorting the intervals and finding the disjoint pairs of intervals, the computation next involves a lot of multiplication of failure predicates. Under our usual *a priori* probability assumption, assume that we want to provide a time service with probability p^d of failure. The criterion for selecting j and k then becomes $\deg(F_{j,k}|FK) \geq d$. By (1), it suffices to calculate FK and $F_{j,k}$ ignoring terms of degree $\deg(FK) + d$ and higher. When we ignore such terms, multiplying m linear failure predicates can be done by a straightforward algorithm in $O(m\mathcal{H} \cdot \mathcal{N}^{\deg(FK)+d-1})$ steps, where $\mathcal{N}$ is the total number of distinct nodes in the predicates and $\mathcal{H}$ is the maximum length of the predicates. (The product is stored as a bit vector of length $\mathcal{N}^{\deg(FK)+d-1}$.) Note that $\deg(FK)$ can be found by computing FK ignoring terms of degree i, for successively larger values of i, until a predicate FK of degree less than i is obtained.

The calculation of $\mathrm{MLM}_{j,k}$ can be done in an obvious manner, where $\mathrm{MLM}_{j+1,k}$ and $\mathrm{MLM}_{j,k+1}$ are calculated using values obtained in the computation of $\mathrm{MLM}_{j,k}$. The computation of $\mathrm{MLM}_{j,k}$, which includes the computation of the $\mathrm{MLM}_{j',k'}$ with $j' < j$ and $k' < k$, takes $O((j+k)\mathcal{H} \cdot \mathcal{N}^{\deg(FK)+d-1})$ steps, which is at most

$$O(\mathcal{H}\mathcal{D} \cdot \mathcal{N}^{\deg(FK)+d-1}) \tag{23}$$

The worrisome factor is $\mathcal{N}^{\deg(FK)+d-1}$. We expect d to be small (perhaps 2); and $\deg(FK)$ should also be small, since it is the number of nodes, among the ones participating in the transmission of the data, that are known to have failed. It seems reasonable for the computational complexity to increase with the number of failures that occur.

If some node A is known to have failed, so $FK = A \cdot FK'$, where FK' does not contain A, then (5) implies that $\deg(F|FK) = \deg(F[1/A]|FK')$. Particular nodes that are known to have failed can therefore be eliminated before performing the calculations. Hence, in (23), $\deg(FK)$ represents the number of known failures for which the failed nodes have not been identified.

5 A Sample Configuration

We now apply these ideas to a simple time service for a network consisting of a (possibly large) number of interconnected LANs. We assume that all the nodes in a LAN can communicate directly with one another in the absence of failures.

Each node contains a *clerk* process that is responsible for providing a time interval to clients on that node. We assume a collection of *time providers,* nodes that are the primary time sources, scattered throughout the network. Each LAN has a collection of *time server* nodes that provide information to the clerks on that LAN. (A node may be both a time provider and a time server.) To simplify the discussion, we assume that each time provider is also a time server. We use the scheme of Section 3.5.4 of partitioning nodes into classes, where time servers are class 1 nodes, and all other nodes are class 2 nodes.

A time server A tries to get time data directly from time providers. Server A then forwards the data it received from each time provider X to the other servers on its LAN. This provides each server B with data having failure predicates of the form X or $X + A$ for a time provider X and a time server A, where A is on the same LAN as B. Normally, there is no reason for B to forward this data to another server C, since C will have received a datum with predicate $X + A$ directly from A. However, such data may be forwarded again by B, giving failure predicates of the form $X + A + B$, if fresh time data are no longer available from provider X (because of communication failure or the failure of X). There are three reasons why B might want to send such a datum to C:

- $C = A$, in which case the reasons for sending the datum back to A were discussed in Section 3.3.1.

- The error rate of B's clock is smaller than that of A's clock.
- Node A cannot transmit its datum again because it has failed. (In fact, the node named C could be the same as the node that used to be named A.)

Assume that there are five servers on a LAN, and these servers get data from a total of five time providers. Suppose that the servers may also have data that originally was relayed by two other servers that have since failed. (Those two could be earlier incarnations of current servers.) A node will have to send a maximum of 35 data, each having a failure predicate of length at most 2, the set of all node predicates containing at most 12 distinct node names. By formula (22) of Section 4.1.2, this gives about 390 bytes of information, excluding the failure knowledge, which is unlikely to require more than 70 bytes.

We now consider the computation performed by a time server to compute the $\mathrm{MLM}_{j,k}$ function before sending data to the clerks on other nodes. A node has at most $5 \cdot 7 \cdot 5$ data with failure predicates of length at most 3. By formula (23) of Section 4.2, to achieve a failure probability of approximately p^d, a time server must perform at most about $525 \cdot 12^{\deg(FK)+d-1}$ calculations. With $\deg(FK)$ equal to 1 and d equal to 2, this is about 1/2 million calculations. This is a very conservative bound, since it assumes that computing $\mathrm{MLM}_{j,k}$ requires examining all of the 175 data, which is extremely unlikely. The actual number is probably closer to 50,000 calculations.

A clerk on a node that is not a time server will obtain data from several time servers on its LAN. If the clerk queried all five servers, it would have to perform at most $50 \cdot (10)^{\deg(FK)+1}$ computations to reduce its 25 data (one for each of the 5 time providers from each server) to a single interval with an $\mathrm{MLM}_{j,k}$ function. Again, this is a rather conservative estimate.

A clerk on a time server node will use the data computed by that server. It will have to further reduce the server's 5 data (one per time provider) to a single interval, using an $\mathrm{MLM}_{j,k}$ function. The clerk does not get additional reduced data from other time servers, since that would violate the principle that class i nodes do not combine data before sending it to other class i nodes. The rationale for this principle is that the clerk need not obtain data from other time servers, since the data it receives from the server on its own node was obtained from data the server obtained from the other servers—the same data that those servers use to compute the data they send to clerks. Another way to view this is that clerks query multiple servers to guard against server failures, but there is little reason for a clerk on the same node as a server to do this. If a server fails, then the clerk on its node has probably also failed.

Acknowledgements

We thank Fred Schneider for suggesting improvements to an earlier version of this note. We also wish to thank the participants at the 1996 Dagstuhl-Seminar on Time Services for their comments.

References

[1] Danny Dolev, Joe Halpern, and H. Raymond Strong. On the possibility and impossibility of achieving clock synchronization. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, pages 504–511, Washington, D.C., 1984. Association for Computing Machinery.

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

[3] Keith A. Marzullo. *Maintaining the Time in a Distributed System*. PhD thesis, Stanford University, February 1984.

[4] Fred B. Schneider. Understanding protocols for byzantine clock synchronization. Technical Report TR87-859, Cornell University, August 1987.