Fault-Tolerant Clock Synchronization in Large Multicomputer Systems

Alan Olson and Kang G. Shin, *Fellow, IEEE*

***Abstract*—The cost of synchronizing a multicomputer increases with system size. For large multicomputers, the time and resources spent to enable each node to estimate the clock value of every other node in the system can be prohibitive. We show how to reduce the cost of synchronization by assigning each node to one or more groups, then having each node estimate the clock values of only those nodes with which it shares a group. Since each node estimates the clock value of only a subset of the nodes, the cost of synchronization can be significantly reduced. We also provide a method for computing the maximum skew between any two nodes in the multicomputer, and a method for computing the maximum time between synchronizations. We also show how the fault tolerance of the synchronization algorithm may be determined.**

***Index Terms*—Clock synchronization, clock skew, clock drift, fault tolerance, multicomputer systems**

I. INTRODUCTION

In a multicomputer system, the cooperation between the nodes is high, often to the point where the system can be thought of as a single computer. In some cases, such as real-time control systems, a systemwide time is used to facilitate cooperation. It determines when each node is supposed to finish certain tasks and when other nodes expect them to finish. Each node measures time with its own clock, and each clock runs at a slightly different rate, and as a result, the difference between each node's idea of the current time increases over time. Such a disparity can cause deadlines to be missed, and eventually lead to system failure.

One possible solution is to reduce the difference between clock rates. Atomic clocks and oven-controlled quartz oscillators are far more accurate than the simple quartz oscillators found in most computers. However, adding such a device to each node can greatly increase the cost, size, weight, and power consumption of the multicomputer. Alternatively, one could use someone else's atomic clock. Universal time coordinated (UTC) can be read via telephone, radio, or satellite from several sources at varying levels of accuracy [2], [16]. Giving each node the hardware necessary to read UTC will make sure the nodes agree on the current time. Once again, the extra hardware will increase the system cost, size, weight, and power consumption. Also, a mobile system may have trouble reading UTC while moving.

A more common solution is to use a synchronization algorithm. All synchronization algorithms have two phases. First, nodes distribute information about their current clock values. Second, each node uses this information to decide how much to adjust its own clock in order to synchronize it with the clocks of the rest of the nodes. Distributing clock information is the costliest part of the synchronization algorithm, with cost generally increasing with accuracy. Since accuracy of synchronization depends directly upon accuracy of clock information, much effort has gone into devising efficient ways to distribute clock information accurately. These methods can be divided into two groups.

Hardware methods [6], [7], [13], [15] use a dedicated network to broadcast each clock signal, and give very accurate results. However, the clock network requires on the order of n^2 communications links for an n node multicomputer, and is thus very expensive for all but the smallest systems. *Network* methods [1], [3], [5], [8]–[12], [14] use the existing communication network to exchange information about clock values. Network methods usually give less accurate results, because of uncertainties in the delays imposed by the communication network, but are much cheaper to implement, because no special hardware is required. A special class of network methods is *probabilistic* methods. They exploit the stochastic nature of communications delays to get very accurate results, at the cost of greater network traffic. Also, accuracy is not guaranteed; instead it has an associated probability, which can be made as close to 1 as is desired.

Many synchronization algorithms require each node distribute information about its clock to every other node. In a large multicomputer, or if tight synchronization is needed, this can be very expensive. Other synchronization algorithms use a master–slave clock organization, so only master clocks need to distribute clock information. This reduces the cost, but creates other problems. The master clock nodes are forced to carry an extra load, and another synchronization algorithm must be provided to synchronize the master clocks.

In this paper, we propose a synchronization algorithm that does not require each node to distribute information about its clock to every other node, and does not require a master–slave clock organization. Instead, each node belongs to one or more groups, and sends information about its clock only to other members of its group. This reduces the cost of synchronization, but does not place the entire load on a few nodes. Also, any of the above methods can be used to distribute

Manuscript received September 1992; revised June 1993. This work was supported in part by Martin Marietta Aeronautics Group in Denver, and by the National Aeronautics and Space Administration (NASA) under Grant NAG-1-1220.

The authors are with the Real-Time Computing Laboratory, Computer Science and Engineering Division, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109–2122 USA; e-mail: alan@eecs.umich.edu, kgshin@eecs.umich.edu.

IEEE Log Number 9403080.

1045–9219/94$04.00 © 1994 IEEE

clock information, so one can be selected on the basis of the accuracy of information provided. Our algorithm also provides a range of synchronization: Nodes in the same group will be tightly synchronized, whereas looser synchronization prevails globally. This reflects the needs of many real-world applications, where cooperating or replicated tasks may need tight synchronization, whereas a looser synchronization may be suitable otherwise.

The paper is organized as follows. Section II presents our assumptions and an overview of our synchronization algorithm. Sections III and IV show how to guarantee a given maximum skew. Section V discusses the fault tolerance of the synchronization algorithm, and how to determine the number of faults it can tolerate. The paper concludes with Section VI.

II. Synchronization

A *clock* is a discontinuous function $C(t)$ mapping from some external real-time (i.e., Newtonian time) reference into the set of integers. Computer clocks are usually nondecreasing functions, because many computers do not like to have their clocks set back; but we do not require this. Each clock has a *drift rate* ρ, such that if $C(t)$ is the clock function and t_1, t_2 are times in the external reference with $t_2 > t_1$, and if the clock is not interrupted or halted in any way during the interval $[t_1, t_2]$, then we have the following equation:

$$(1-\rho)(t_2 - t_1) \leq C(t_2) - C(t_1) \leq (1+\rho)(t_2 - t_1). \quad (2.1)$$

For a good crystal oscillator, ρ will be on the order of 10^{-6}. In real clocks, the value of ρ is not constant, but changes over time. Because the change almost always happens very slowly, we can neglect its effects.

Two clocks, $C_i(t)$ and $C_j(t)$, are said to be *δ-synchronized* at time t if and only if:

$$|C_i(t) - C_j(t)| \leq \delta. \quad (2.2)$$

The purpose of a synchronization algorithm is to ensure that any pair of clocks in the multicomputer will be δ-synchronized whenever the multicomputer is in operation. Synchronization algorithms that use hardware methods to distribute clock information usually run continuously, constantly adjusting clock rates to keep clocks synchronized. Other algorithms, including the one we describe here, run periodically, and clocks can drift apart between runs of the synchronization algorithm. Such algorithms generally synchronize clocks to within some *target* $\tau < \delta$. The value of τ is chosen to be small enough so that clock drift cannot increase skew to more than δ before the next run of the synchronization algorithm, i.e., if T is the real time that elapses between successive synchronization:

$$T \leq \frac{\delta - \tau}{2\rho}. \quad (2.3)$$

A. Assumptions

We make the following assumptions about the multicomputer, the clocks, and the way clock information is interpreted.

1) Each node of the multicomputer has its own hardware clock. The clock consists of a counter whose value can be read at any time.
2) A method exists to allow each node to gather information about the clocks of other nodes in the multicomputer. We do not require each node be able to gather information about all other nodes, only a subset sufficient to ensure synchronization, as detailed in Section III.
3) The value of the clock *increases* at an almost constant rate over the period where clock information is being distributed through the multicomputer. The frequency of all real clocks varies somewhat, and also the synchronization algorithm may insert or suppress clock cycles periodically in order to synchronize the clock. This is tolerable as long as the variation in rate is small in comparison to the rate itself.
4) Each node uses the information it gathers to estimate the difference, or *skew*, between its clock and the clocks of other nodes. The error of any given estimate is no more than ϵ. This accuracy may be guaranteed, or may have an associated probability if probabilistic methods are used.

These assumptions allow any of the distribution methods mentioned above to be used with little or no modification. Hardware distribution methods usually connect lines from other node's clocks to a phase-locked loop, but for our purposes, each would be connected to a counter, and the difference in counter values would give the skew between clocks. Network methods require clocks to be counters anyway, and all synchronization algorithms that use them produce skew estimates either explicitly or implicitly.

B. Overview of the Proposed Algorithm

The amount of clock information to be distributed can be reduced if each node is selective about the other nodes from which it gathers information. It is possible to provide synchronization while requiring each node only gather clock information from a few carefully selected nodes. This is the basis of our algorithm.

We begin by defining sets of nodes, called *synchronization groups*. We guarantee the clocks of any two nodes that belong to the same synchronization group will be δ-synchronized. Overlap between synchronization groups guarantees that if two nodes do not belong to the same synchronization group, there exists some integer k such that the skew between the clocks of the two nodes is less than $k\delta$.

Each node belongs to at least one synchronization group, and gathers clock information from, and sends clock information to, only those nodes with which it shares a synchronization group. A node uses the clock information that it receives from another node to estimate the skew between its clock and the clock of the other node. An interactive convergence algorithm [8] is then applied. Any skew estimates that are too large are discarded; the rest are averaged, and the result is the amount by which the clock is to be adjusted. Large skews are discarded

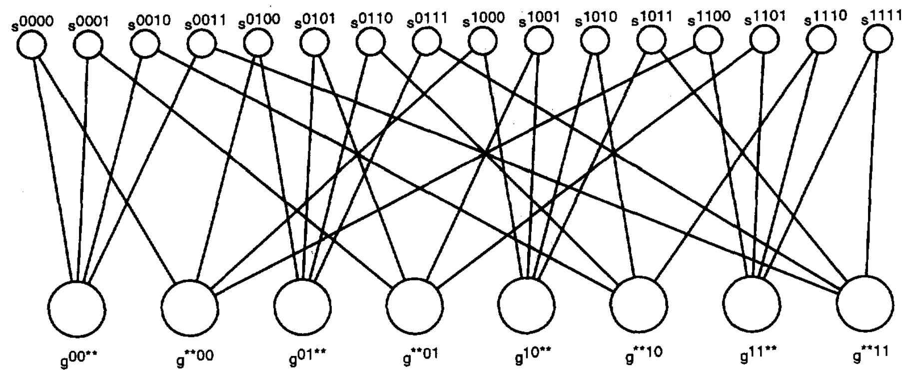


Fig. 1. Synchronization graph for a 16-node hypercube multicomputer.

to present a single faulty node from throwing off the average, a skew α is considered too large if $|\alpha| > \delta + \epsilon$ (the maximum skew plus any estimation error). A single application of this algorithm will synchronize the clocks of a synchronization group to within τ, and the algorithm must be run again before the clocks can drift to more than δ apart.

The guarantees of maximum skew depend on the assignment of nodes to synchronization groups. If there is not enough overlap between groups, the guarantees will not hold. In Section III and IV, we show how to determine if a given set of synchronization groups will provide the necessary guarantees. Section III presents a graph-oriented method for describing synchronization groups, and Section IV shows how to use the graph to compute τ.

C. Initial Synchronization

Any guarantees of maximum skew also depend on the method of achieving initial synchronization. This would normally be done during system initialization. Any algorithm for achieving initial synchronization [5], [9] can be used. Additionally, a temporarily appointed master could repeatedly broadcast its clock value, much as in [1], until the rest of the nodes can get an accurate-enough estimate. If none of these approaches can synchronize the multicomputer tightly enough, repeated applications of our algorithm can be used to finish the job. If the nodes of synchronization group are synchronized to within Δ, a single application or our algorithm will reduce the skew to some $\tau_1 < \Delta$. Immediately applying our algorithm again will result in a maximum skew of $\tau_2 < \tau_1$. Continuing in this fashion will eventually result in a maximum skew of $\tau_f \leq \tau$, the target value during normal operation.

III. Synchronization Groups

Each node will belong to at least one synchronization group. Two nodes are said to be *tied* if they are both members of the same synchronization group. The transitive closure of tied is *strung,* e.g., nodes A and B are strung, because A is tied to C, which is tied to D, which is tied to B. Because there is a limit on the skew between tied nodes, there will exist a limit on the skew between strung nodes. If two nodes are not strung, there will be no limit on the skew between them. To ensure synchronization, a node need not be tied to every other node, but it must be strung to every other node.

As a simple example, consider a 16-node hypercube. Each node will have a binary address between 0000 and 1111. We define eight synchronization groups of four members each. The first four groups will be the subcubes of the form $ab**$, and the second four groups will be subcubes of the form $**ab$, where $*$ indicates a "don't care" address bit and $a, b \in \{0, 1\}$. Every node will belong to two synchronization groups, each with three other nodes. Thus, each node will be tied to six other nodes. Any pair of nodes will be strung.

A. The Synchronization Graph

A bipartite *synchronization graph* can be derived from the definitions of the synchronization groups. One vertex set, called the *group vertices,* has one vertex for each synchronization group; the other vertex set, called the *system vertices,* has one vertex for each node. The system vertex that corresponds to node a is denoted as s^a, and the group vertex that corresponds to synchronization group A is g^A. An edge exists between s^a and g^A if and only if node a is in synchronization group A. Nodes a and b are tied if and only if the distance between s^a and s^b is 2. Nodes a and b are strung if and only if there is a path between s^a and s^b. If the synchronization graph is connected, then all pairs of nodes will be strung. Throughout this paper, we use *nodes* to refer to the physical nodes of the multicomputer, and *vertices* to refer to vertices of the synchronization graph.

Fig. 1 shows the synchronization graph for the 16-node hypercube example discussed in the previous section, where the small circles represent system vertices and large circles are group vertices.

The *stretch* between a pair of nodes is *half* the length of the shortest path between their respective vertices in the synchronization graph; e.g., the stretch between a pair of tied nodes is 1. In Fig. 1, the distance between any two system vertices is no greater than 4, so the stretch between any two nodes in this multicomputer is no greater than 2. The stretch between a pair of nodes multiplied by δ yields the maximum skew between the nodes. 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 from Fig. 1, where the maximum stretch is 2, but the diameter of a 16-node hypercube is 4.

B. Synchronization Paths

We call the set of nodes tied to a given node that node's *synchronization set*. For example, in our 16-node hypercube example, the synchronization set of node 0000 will be $\{0001, 0010, 0011, 0100, 1000, 1100\}$. When a node computes the adjustment to its clock, it will use the estimated skews for all nodes corresponding to vertices in its synchronization set. Some of these nodes may not belong to a common synchronization group; e.g., in the earlier example, 0000 will use skew estimates from 0011 and 1100 that do not belong to a common synchronization group. It would therefore be impossible for 0000 to remain synchronized with both 0011 and 1100 if there were no bounds on the skew between the latter two. The stretch provides the needed bound, and it can be computed from the synchronization graph. However, when fault tolerance is considered, we need to know more than the stretch. We need to know upon which nodes and synchronization groups the stretch depends. Specifically, for any pair of nodes, we need to know what paths exist between their respective nodes in the synchronization graph.

In a multicomputer with synchronization graph S, node a and corresponding system vertex s^a, let S^a be the set of all vertices a distance of 2 from s^a (i.e., vertices corresponding to node a's synchronization set). We then define *synchronization paths* (SP's) for a and b as follows.

Definition 1: Given a multicomputer with synchronization graph S, and tied nodes a and b, let $S_\cap^{ab} = S^a \cap S^b$, and let G^{ab} be the set of group vertices a distance of 1 from exactly one of s^a and s^b. A synchronization path (SP) is a simple path in S from a member of S^a to a member of S^b, which contains at most one member of $S_\cap^{ab}$, no members of G^{ab}, and has length no greater than 4.

The existence of an SP between $s^d \in S^a$ and $s^e \in S^b$ indicates a relationship between node d in a's synchronization set and node e in $b's$ synchronization set. Either d and e are tied to each other, or are both tied to some common third node.

As an example, consider Fig. 1 again. Nodes 0000 and 0001 are both in synchronization group $00**$. We have:

$$\begin{aligned} S^{0000} &= \{S^{0001}, S^{0010}, s^{0011}, s^{0100}, s^{1000}, s^{1100}\} \\ S^{0001} &= \{S^{0000}, S^{0010}, s^{0011}, s^{0101}, s^{1001}, s^{1101}\} \\ G^{0000\,0001} &= \{G^{**00}, g^{**01}\}. \end{aligned}$$

Each of the members of S^{0000} not in $S_\cap^{0000\,0001}$ has an SP of length 2: $s^{0100} \to g^{01**} \to s^{0101}$, $s^{1000} \to g^{10**} \to s^{1001}$, and $s^{1100} \to g^{11**} \to s^{1101}$. There are also 12 SP's of length 4.

Each SP has a corresponding stretch of half the length of the SP. Each SP is classified according to its corresponding stretch. 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*.

We can find all SP's for a given pair of nodes using a simple modification of a depth-first search algorithm. Procedure *find_synch_path*, shown in Fig. 2, will find all SP's that 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^a 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.

```
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
```

Fig. 2. Procedure *find_synch_path*.

C. Synchronization Areas

To show that the multicomputer will remain synchronized, we must show that immediately after the synchronization algorithm has finished, the skew between any pair of tied nodes will be less than τ. This could require checking a large number of cases. Fortunately, it is not necessary to check most of the cases.

Definition 2: Given synchronization graph S and tied nodes a and b, the *synchronization area* of a and b is a subgraph of S containing s^a, s^b, all group vertices a distance of 1 from s^a and s^b and the edges connecting them, all system vertices a distance of 1 from these group vertices and the edges connecting them, and all vertices and edges contained in SP's for a and b.

The synchronization area of s^a and s^b is all vertices and edges contained in paths of length less than or equal to 8 between s^a and s^b. This will contain all SP's for vertices in S^a and S^b. Because the SP's determine the maximum skew between nodes a and b, 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 that 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., if one can be transformed to the other by simply relabeling its vertices, then they are said to be *equivalent*. To show that the multicomputer is synchronized, one has to show synchronization for all possible nonequivalent synchronization areas; i.e., any synchronization areas equivalent to synchronization areas already checked do not have to be checked. This greatly reduces the number of cases. Often, as in Fig. 1, the synchronization graph will be identical from the point of view of any system vertex. More specifically, if a is a node, a labeling of the vertices of the

synchronization graph can be found that gives the label s^a to any desired system vertex. In Fig. 1, the label s^{0000} has been given to the system vertex at the far left; it could just as easily been given to the system vertex at the far right, or to any system vertex in between. In such cases, all synchronization areas are equivalent.

IV. Maximum Skew

When nodes a and b synchronize, each will estimate the clock skew between itself and every node in its synchronization set. Each will then average the resulting skew estimates to get the amount by which to adjust its own clock. If a and b are in the same synchronization set, we want the clock skew between them *after* they have adjusted their clocks (assume that clocks are adjusted instantaneously) to be less than the target skew, τ. It is trivial to show that this is true in standard synchronization algorithms, where there is a single synchronization group containing all nodes. In our case, things are more complicated, and in this section, we show how to prove that the multicomputer will remain synchronized. It should be understood that we are showing only how to *prove* the algorithm works, not describing the algorithm's operation. None of the computations done in this section have to be made by the synchronization algorithm while it is operating.

Assume that nodes a and b are tied. Let n^a and n^b be the number of skew estimates made by a and b; i.e., n^a is the number of nodes in a's synchronization set, plus one (the extra one is the skew for node a itself, which is always 0). Let T^a and T^b be the *sum* of the skew estimates computed by a and b. Without loss of generality, we can assume that b has a greater clock value than a. At worst, the skew between a and b is already the maximum allowable, δ. The maximum skew between a and b after synchronization can then be found by maximizing the following quantity:

$$\delta + \frac{T^b}{n^b} - \frac{T^a}{n^a}. \tag{4.1}$$

In order to show synchronization, it must be less than τ, i.e., as follows:

$$\tau \geq \delta + \max\left(\frac{T^b}{n^b} - \frac{T^a}{n^a}\right). \tag{4.2}$$

Consider the synchronization area of a and b. Assume that the skew of b with respect to a is δ. We wish to give skews, with respect to the appropriate node, to each vertex in S^a and S^b, so that if these were the skews computed for their respective nodes, the value in (4.1) would be maximized. At worst, members of S^b are given skews of δ with respect to b, and members of S^a are given skews of $-\delta$ with respect to a. This implies a skew of 3δ between members of S^b and S^a. But some vertices will be in both S^a and S^b, 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 δ between its endpoints, and a 2-SP indicates a maximum skew of 2δ between its endpoints. By giving a skew to one vertex, we limit the skews that can be given to a number of other vertices, and by giving skews to these vertices, we limit the skews we can give to even more vertices, and so on. Finding the maximum skew between a and b after synchronization is therefore a process of searching a large number of cases.

We need to be able to compute the maximum quickly. To do this, we break the problem 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.

A. Clusters

The problem is one of maximizing a quantity subject to certain restrictions. We err on the safe side if we ignore some restrictions, because removing restrictions will not reduce the maximum. The SP's correspond to the restrictions, and we simplify the problem by eliminating many of the SP's. We partition the members of S^a and S^b into *clusters*. A cluster is a group of vertices where each vertex has an SP to at least one other vertex in the cluster. We ignore any SP's between clusters (and many SP's within a cluster), allowing us to consider 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^a and S^b can be typed by the length of the shortest SP for which it is an endpoint. A vertex in $S^{ab}_{\cap}$ is an *intersection* vertex, a vertex that is the endpoint of a 1-SP is a *1-vertex*, a vertex that is the endpoint of a 2-SP (but no 1-SP's) is a *2-vertex*, and a vertex that 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^d is to be assigned to vertex s^c, the following two requirements must be met.

1) There must be an SP with endpoints at s^d and s^c.
2) If s^d is a 1-vertex, there must be a 1-SP with endpoints at s^d and s^c. Note that s^c then must be either a 1-vertex or an intersection vertex.

Because each assignment corresponds to some SP, either $s^d \in S^a$ and $s^c \in S^b$, or the reverse. Also, notice that a 1-vertex must be assigned to either a 1-vertex or an intersection vertex, whereas a 2-vertex can be assigned to either a 2-vertex, a 1-vertex, or an intersection vertex.

A cluster is a minimal nonempty set of vertices such that for every vertex s^d in the cluster, the cluster contains all vertices assigned to s^d, and the vertex to which s^d 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^d is assigned to s^c, and if some vertex s^e is assigned to s^d, 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^d is assigned to s^c and s^c already belongs to a cluster, then s^d belongs to s^c's cluster.
3) If s^d is assigned to s^c and s^c does not belong to any cluster, a new cluster is created with s^c and s^d as members. Furthermore, if s^c and s^d are the same type vertices (i.e., both 1-vertices or both 2-vertices) assign s^c to s^d. Notice that if s^c is a 1-vertex, then s^d 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, we must be careful when selecting assignments. If vertex s^d is to be assigned, then for each s^c to which s^d could be assigned, place s^c in whichever of the following sets in appropriate:

1) Vertices of the same type as s^d that do not belong to a cluster.
2) Vertices of the same type as s^d that belong to a straight cluster.
3) Intersection vertices that do not belong to a cluster.
4) Vertices that belong to a jumbled cluster.
5) Vertices of a different type than s^d that belong to a straight cluster.

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

B. Computing Skew Terms

Clusters allow us to reduce the maximizing problem of (4.2) to one of maximizing a sum of terms. The terms are formed by breaking up the T^a and T^b sums and reorganizing and mixing pieces to form terms of the form $\frac{x}{n^b} - \frac{y}{n^a}$. 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 that are in S^b, and y as the sum of skews for vertices in the cluster that are in S^a. There will be one term for each cluster, one term for estimation error, one term for the skew between a and b, one term for the unbound vertices, and one term for the intersection vertices that 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 SP's between clusters we are ignoring), the maximum of the sum will be an upper bound on the actual maximum skew between a and b after synchronization.

The maximum of each term is the maximum contribution that each term may make to the skew between a and b after synchronization. In most cases, finding the maximum means checking several possible worst-case configurations of vertices to see which is the maximum. Section IV-B 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 IV-C.

Noncluster Terms: The error term represents the maximum contribution to the skew because of the inaccuracy of the estimation process. If the maximum error is ϵ, estimation error can subtract $\epsilon(n^a-1)$ from T^a and add $\epsilon(n^b-1)$ to T^b. (There is no error in estimating one's own clock.) The value of this term is then $\epsilon\frac{n^b-1}{n^b} + \epsilon\frac{n^a-1}{n^a}$.

The next term is generated by each node estimating the other's clock. If the skew between them is δ, and b has the greater clock value, the value of this term is $-\delta/n^b - \delta/n^a$. This term will always reduce the maximum skew.

Because the unbound vertices have no SP's, their skew values are bound only because they belong to either S^a and S^b. Therefore, as a worst case, the unbound vertices in S^a are given skews $-\delta$, and the unbound vertices in S^b are given skews δ. If there are $n^{a,u}$ unbound vertices in S^a and $n^{b,u}$ unbound vertices in S^b, this term has value $\delta\left(\frac{n^{b,u}}{n^b} + \frac{n^{a,u}}{n^a}\right)$.

Intersection vertices will be given skews relative to both a and b. These skews must be consistent; i.e., for a given vertex, the skew with respect to a must be δ greater than the skew with respect to b. If there are $n^{\cap,u}$ intersection vertices that do not belong to any cluster, and if the total skew with respect to a of these vertices is $T^{\cap,u}$, then the value of this term is $\frac{T^{\cap,u}-\delta n^{\cap,u}}{n^b} - \frac{T^{\cap,u}}{n^a}$. This value is not constant, but is a function of $T^{\cap,u}$. Because the intersection vertices must remain within δ of both *a and b*, their skews with respect to a must be in $[0,\delta]$. This gives $T^{\cap,u}$ a range of $[0,\delta n^{\cap,u}]$. The function is linear, and so will have its maximum at one of the endpoints. The maximum value of this term is then the maximum of $-\delta\frac{n^{\cap,u}}{n^b}$ and $-\delta\frac{n^{\cap,u}}{n^a}$. This term also will reduce only 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^a as far from the vertices of S^b as the SP's will allow, while keeping their skews less than δ. This is similar to the original problem of giving skews to the vertices we discussed at the beginning of Section IV, only now we have greatly simplified matters by considering only one cluster at a time and by ignoring SP's between clusters. We further simplify by considering only those SP's within a cluster that 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 that could yield a maximum value for its term.

Number the clusters from 1 to C, where C is the total number of clusters. We define $n_i^{a,1}$ to be the number of 1-vertices from S^a that are members of cluster $i, n_i^{a,2}$ to be the number of 2-vertices from S^a in cluster i, and similarly for vertices in S^b. In a similar fashion, we define $S_i^{a,1}, S_i^{a,2}, S_i^{b,1}$ and $S_i^{b,2}$ to be the sets of vertices in cluster i.

The preference for assigning vertices to unassigned vertices has an interesting consequence for straight clusters of 1-vertices. Every vertex in $S_i^{a,1}$ will be assigned to the same

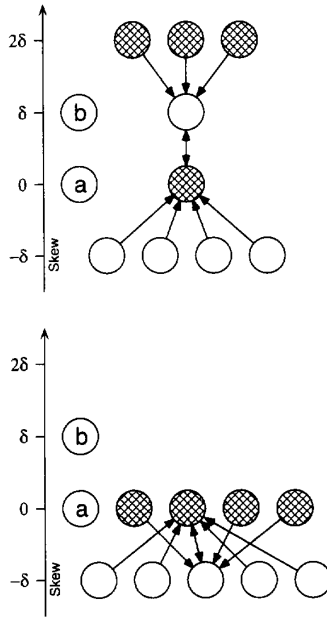


Fig. 3. Configuration for a straight cluster of 1-nodes.

vertex in $S_i^{b,1}$ (the *point* of $S_i^{b,1}$), and every vertex in $S_i^{b,1}$ will be assigned to the same vertex in $S_i^{a,1}$ (the point of $S_i^{a,1}$). 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 $S_i^{a,1}$ and the point of $S_i^{b,1}$ are given skews $-\delta$, whereas all but the point of $S_i^{b,1}$ and the point of $S_i^{a,1}$ are given skews δ. Fig. 3 shows the two configurations: the fractured configuration on top, and the whole configuration on the bottom. The line at left shows skews with respect to a. Subtracting δ from this value gives skew with respect to b. The vertices of $S_i^{b,1}$ are cross-hatched, and assignments are indicated by the arrows. The whole configuration always yields the maximum value when either $S_i^{a,1}$ or $S_i^{b,1}$ 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 $-\delta$ or δ. The maximum skew for the whole configuration is then the maximum of $\delta\left(\frac{n_i^{a,1}}{n^a} - \frac{n_i^{b,1}}{n^b}\right)$ and $\delta\left(\frac{n_i^{b,1}}{n^b} - \frac{n_i^{a,1}}{n^b}\right)$. The fractured configuration usually yields the maximum when both $S_i^{a,1}$ and $S_i^{b,1}$ have more than one member. This configuration gives a skew of $\delta\left(\frac{n_i^{b,1}-2}{n^b} + \frac{n_i^{a,1}-2}{n^a}\right)$. The larger of the skews for the two configurations is the value of the term.

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, because nodes may now be 2δ apart. The whole configuration yields values $\delta\frac{n_i^{a,2}}{n^a}$ and $\delta\frac{n_i^{b,2}}{n^b}$. The fractured configuration yields a value of $\delta\left(\frac{n_i^{b,2}-1}{n^b} + \frac{n_i^{a,2}-1}{n^a}\right)$.

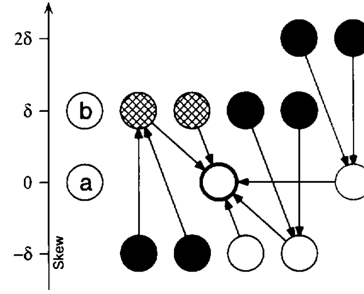


Fig. 4. Configuration for a jumbled cluster without an intersection vertex.

Jumbled clusters can be one of two kinds: Either they contain an intersection vertex or they do not. We consider each separately.

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. Whole and fractured configurations exist just as they do in straight clusters. The main difference is the extra consideration that must be given to the skew given to the 2-vertices and the 1-vertices to which they have been assigned. In both configurations shown in Fig. 4, a pair of 2-vertices (gray-filled) are assigned 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. It is helpful to consider each of these groupings as a subcluster separate from the rest of the cluster. As an example, Fig. 4 contains only one subcluster, the one discussed above. It contains three vertices, and its value is the maximum of $-\delta$ and 2δ (the two possible terms generated by the subcluster). With some modification for subclusters, computation of maximum term value proceeds much like it does for straight clusters.

The fractured configuration in the upper half of Fig. 4 is handled just like the analogous arrangement for straight clusters, except that the subclusters are handled separately. The term is computed as if the subcluster vertices did not belong to the cluster; then the subcluster terms are added in. The whole configuration is somewhat more complex, because it will now have three possible maxima. The first two are the same as for the straight cluster; the nonsubcluster vertices will be given skews $-\delta$ or δ. When they are given skews $-\delta$ (as shown in the lower part of Fig. 4), subclusters containing vertices in $S_i^{a,1}$ must be considered separately, whereas when they are given skews δ, subclusters containing vertices in $S_i^{b,1}$ must be considered separately. The third possible maximum is brought about by the presence of 2-vertices. Members of $S_i^{a,1}$ and $S_i^{b,2}$ will have skews 0 and $-\delta$, whereas members of $S_i^{b,1}$ and $S_i^{b,2}$ will have skews 0 and δ, and subclusters are not considered. This generates a value of $\delta\left(\frac{n_i^{b,2}}{n^b} + \frac{n_i^{a,2}}{n^a}\right)$. The largest of the values becomes the value of the cluster term.

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, will be assigned to the sole intersection vertex. The 2-vertices

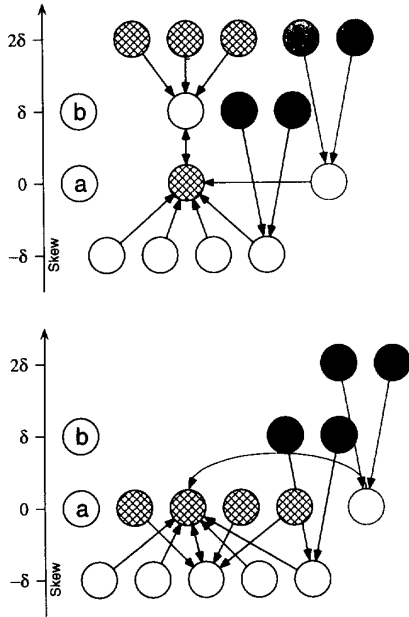


Fig. 5. Configuration for a jumbled cluster containing an intersection vertex.

assigned to 1-vertices will form subclusters, as in the case of jumbled clusters without intersection vertices. Once again, we get 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 δ with respect to a. The configuration where it is given 0 is shown in Fig. 5, and the intersection vertex is shown with a bold border. In this configuration, any subclusters containing vertices of $S_i^{a,1}$ must be considered separately. If there are $n_i^{a,1f}$ vertices not in any subcluster in $S_i^{a,1}$, and $n_i^{b,2f}$ vertices not in any subcluster in $S_i^{b,2}$, then this configuration yields value $\delta\left(\frac{n_i^{b,2f}-1}{n^b}+\frac{n_i^{a,1f}+n_i^{a,2}}{n^a}\right)$, plus any subcluster values. If the intersection vertex is given skew δ with respect to a, then any subcluster containing vertices of $S_i^{b,1}$ must be considered separately. If there are $n_i^{b,1f}$ vertices not in any subclusters in $S_i^{b,1}$, and $n_i^{a,2f}$ vertices not in any subcluster in $S_i^{a,2}$, then this arrangement produces value $\delta\left(\frac{n_i^{b,1f}+n_i^{b,2}}{n^b}+\frac{n_i^{a,2f}-1}{n^a}\right)$, plus any subcluster values. The largest value yielded by these configurations becomes the value for this cluster term.

C. Relating $\epsilon, \delta,$ and τ

If synchronization is to be maintained, we must show that the value in (4.1) is less than or equal to τ, i.e., we have:

$$\tau \geq \delta + \frac{T^b}{n^b} - \frac{T^a}{n^a}. \tag{4.3}$$

From the results of Section IV-B, we know that T^a and T^b are functions of ϵ and δ. So, (4.3) relates three variables, $\tau, \epsilon,$ and δ. If we know the value of one of them, we can solve to get a minimum ratio for the remaining two; e.g., if ϵ is 1 ms, we can substitute into (4.3) to get a minimum ratio between τ and δ. Even if we know only the ratio of two of the variables, we can determine the minimum ratio for the other two; e.g., if $\delta = 10\epsilon$, we can substitute into (4.3) to get a minimum ratio between τ and δ.

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

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

If $\delta = 10\epsilon$, then $\tau \geq \frac{3}{5}\delta$, or $\tau \geq 6\epsilon$. A good estimation algorithm might have an ϵ of 1 ms; then δ is 10 ms, and τ is at least 6 ms. If we assume a clock drift ρ of 10^{-6}, (2.3) gives a time between synchronization of no more than 4000 s.

D. 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 SP's. 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 that dominates the complexity of computing maximum skew.

In practice, cluster assignment is very fast. All SP's 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, regardless of whether it contains an intersection vertex, the numbers of nodes of each type) can be derived as the cluster is formed; only subclusters require special handling.

E. Defining Synchronization Groups

Although we have defined what a synchronization group is, and have shown how to compute the maximum skew for them, we have said nothing about how to define them for a particular multicomputer. For some multicomputers, 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 multicomputers. 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 that can be used to generate good synchronization groups for most systems. For an n-node multicomputer, lay out the nodes in a $\sqrt{n} \times \sqrt{n}$ grid, filling any empty spaces with dummy nodes. Defining each row and each column of the mesh 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.

V. Fault Tolerance

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

A. Fault Model

Any analysis of fault tolerance depends upon the fault model used. Since we make few assumptions about the way in which clock information is distributed, we assume that estimates are *trustworthy*. That is, whenever any nonfaulty node estimates its clock skew with respect to another nonfaulty node, that estimate will be accurate to within ϵ. Put another way, no faulty node can alter or destroy clock information sent out by nonfaulty nodes in a way that cannot be detected and corrected. This may be accomplished by the network through digital signatures or multiple copies of messages, or by other means; or it may be provided by the distribution method. We can then model faults by removing components of the synchronization graph. We have the following fault types.

- *Node Faults:* This corresponds 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:* This corresponds to the removal of an edge from the synchronization graph. Communication failures often fall into this type. Because these faults have effects less than the node faults of their endpoints, we do not consider them.
- *Group Faults:* This corresponds to the removal of a group vertex from the synchronization graph. Faults that prevent the clock information from getting distributed 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 we can no longer guarantee maximum skew between a pair of tied nodes.

A group fault that is caused by the presence of node faults is called an *induced* fault. There is nothing wrong with the synchronization group itself; its nonfaulty members may continue to estimate each other's clock values, but the guarantee of a maximum skew of δ between members no longer holds.

B. Determining Fault Tolerance

We consider the multicomputer 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. We consider each of these problems separately.

Connectedness of the Synchronization Graph: Because each fault corresponds to the removal of a component of the synchronization graph, multiple faults may disconnect the graph. The number of faults that the multicomputer can withstand is therefore limited by the minimum number of faults that 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 $O(EV\log(V^2/E))$ [4].

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 nonfaulty members, as calculated in Section IV, is greater than τ. The collapse of a synchronization group is an induced group fault.

The problem with induced faults is that 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 we can 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, and then check each to find the minimum number of faults to cause a synchronization group to collapse. These are the same areas that 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 τ, or have small values of either n^a or n^b.

Finding the minimum fault set to collapse a synchronization group requires searching all fault sets of the synchronization area. Although 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 that

we took was 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 with a depth of 1 in the tree correspond to fault sets of size 1, nodes a depth of 2 correspond to fault sets of size 2, and so on. Each node will also have 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 that has a maximum skew greater than δ. Our implementation used 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 that 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, because 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. We had two strategies for doing this. The first strategy was to rate each fault set according to the types of faults that it contained. Fault sets that 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. We quickly abandoned the first strategy, because it would take days to find sets that 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 though 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 our search found the minimum fault set quickly, a large number of nodes may still have to be searched. In one of our examples below, the synchronization area will contain in excess of 40 system vertices, whereas 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. But there are over 5 billion of these nodes. To allow our search to finish within a reasonable amount of time, we employed two strategies that 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 who do not have a greater maximum skew than their parent. These strategies greatly reduced the search time, from 22 days to less than 24 hr in one case.

Any strategies to reduce the number of nodes searched may cause the search to overlook the minimum fault set. We encountered no such cases (in fact, the first set found was almost always the minimum set), but it would be wise to consider the smallest set found to be only an estimate of the actual fault tolerance. As we show in the examples, it is possible to intentionally underestimate the fault tolerance by changing ϵ and τ. This can be used to reassure oneself that the multicomputer will have the desired fault tolerance. It should also be possible to intentionally underestimate fault tolerance by ignoring 2-SP's. This should speed up the search (though we did not try it), and could be used for multicomputers where the minimum fault set is too large for the search to find within a reasonable time.

C. Examples

Faults have the effect of increasing the value in (4.1). A τ that works when there are no faults may be too small when faults are present. τ must be chosen somewhat larger than the value of (4.1) in order to satisfy (4.3) when faults are present. In each example, we must consider only a single synchronization area, because the synchronization graphs are such that any two pairs of tied nodes will have isomorphic synchronization areas. This is partly because of the homogeneous nature of the multicomputers that we consider, and partly because such graphs are easier to deal with.

We start with the 16-node hypercube of Fig. 1. For a best-case estimate of fault tolerance, we assume no estimation error and continuous synchronization; i.e., $\epsilon = 0$ and $\tau = \delta$. In order to collapse a synchronization group, a minimum of five node faults or three group faults is needed. fault tolerance decreases as ϵ increases, and it increases as τ increases. If we increase ϵ to $.1\delta$ and decrease τ to $.9\delta$, either three node faults or one group fault will collapse a synchronization group. Thus, three node faults may induce a cascade of group faults that will engulf the multicomputer.

For our second example, we consider a 16×16 square mesh, wrapped on the edges to provide a homogeneous multicomputer. We use the method suggested in Section IV-E to define 32 synchronization groups of 16 members each, one group for each row of the mesh, and one group for each column. If we let $\epsilon = 0$ and $\tau = \delta$, either 14 node faults or nine group faults are needed to collapse a synchronization group. We can increase ϵ and decrease τ to the more realistic values of $\epsilon = .1\delta$ and $\tau = .9\delta$, but at the cost of some fault tolerance. In this case, either six node faults or one group fault are needed to collapse a synchronization group. So, at least six 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. We consider a 256-node hypercube with 64 synchronization groups of 16 members each. The hypercube has an 8-bit address, $abcdefgh$. Each synchronization group will be 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 we let x indicate a "don't care" position in the address, we can define the 64 subcubes 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 $\epsilon = 0$ and $\tau = \delta$, the search algorithm did not terminate in over two weeks. The smallest set found was 27 node faults. We can get the search to terminate by increasing ϵ and decreasing τ. When $\epsilon = .1\delta$ and $\tau = .8\delta$, either 13 node faults or two group faults are needed

to collapse a synchronization group. We can reduce τ further and still tolerate a fair number of faults. If $\tau = .8\delta$ then eight node faults are required to collapse a synchronization group.

Last, we consider two 1024-node multicomputers, a 10-cube, and a 32×32 wrapped square mesh. For the 10-cube, we define 64 synchronization groups, thirty-two 5-cubes of the form $abcdexxxxx$, and thirty-two 5-cubes of the form $xxxxxfghij$. For the square mesh, we again define each row and each column to be a synchronization group. Each of these two multicomputers will have 64 groups of 32 members each. In fact, the synchronization graphs for the two multicomputers are isomorphic, so their fault tolerances are identical. These multicomputers are too large to solve easily for $\epsilon = 0$ and $\tau = \delta$. If we increase ϵ to $.1\delta$ and decrease τ to $.9\delta$, we find each multicomputer requires 10 node faults or one group fault before a synchronization group can collapse. We can further reduce τ to $.8\delta$ and still require five node faults. Although a set of 10 faults may not seem like much in a 1024-node multicomputer, 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 multicomputer. Also, fault tolerance can be improved by increasing the number of groups, as we showed in the previous example.

The above examples clearly show the ability of our algorithm to synchronize large multicomputers while reducing synchronization overhead. The reduction in overhead 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 six nodes, so each node needs to communicate with only six other nodes in order to synchronize. This is compared with the 15 nodes of standard algorithms, a twofold reduction. For the 16×16 square mesh, we get a reduction of 255/30, or more than eightfold; doubling the number of synchronization groups to improve fault tolerance gives us a savings of only fourfold. In our last example, each node needs to communicate with only 62 out of 1023 other nodes in order to synchronize, which is more than a 16-fold decrease in synchronization overhead.

Our 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 τ, 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 tolerance that the search would take too long to find.

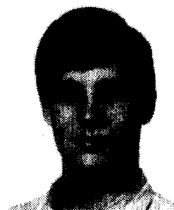
VI. Conclusion

In this paper, we have presented an algorithm that greatly reduces synchronization overhead for large multicomputers by reducing the amount of clock information that has to be distributed. The nodes of the system are assigned to groups, and each node distributes information about its clock only to the nodes with which it shares a group. Our algorithm works with many different clock estimation algorithms, so one may consider the trade-offs of overhead versus accuracy that come with different algorithms. The algorithm also provides a natural way to map real-time tasks into the system, in that parts of the system have the tight synchronization needed by cooperating or replicated tasks while the system as a whole still remains synchronized.

We presented a method for analyzing the algorithm, and determining the maximum skew. The method can also be used to determine the fault tolerance. We used this method to analyze the fault tolerance of several systems, including a 1024-node hypercube. We also showed how the fault tolerance of a system can be adjusted by changing the number of groups, and presented a simple method for defining synchronization groups that works well in many cases.

References

[1] K. Arvind, "A new probabilistic algorithm for clock synchronization," in *Proc. Real-Time Syst. Symp.*, 1989, pp. 330–339.
[2] R.E. Beehler and D.W. Allan, "Recent trends in NBS time and frequency distribution services," *Proc. IEEE*, vol. 74, no. 1, pp. 155–157, Jan. 1986.
[3] F. Cristian, "Probabilistic clock synchronization," *Distrib. Computing* , vol. 3, pp. 146–158, 1989.
[4] A.V. Goldberg and R.E. Tarjan, "A new approach to the maximum flow problem," in *Proc. 18th Ann. ACM Symp. on Theory of Computing*, 1986, pp. 136–146.
[5] J.Y. Halpern, B. Simons, R. Strong, and D. Dolev, "Fault-tolerant clock synchronization," in *Proc. 3rd Symp. Principles Distrib. Computing*, 1984, pp. 89–102.
[6] J.L.W. Kessels, "Two designs of a fault-tolerant clocking system," *IEEE Trans. Comput.* vol. C-33, no. 10, pp. 912–919, Oct. 1984.
[7] C.M. Krishna, K.G. Shin, and R.W. Butler, "Ensuring fault tolerance of phase-locked clocks," *IEEE Trans. Comput.* vol. C-34, no. 8, pp. 752–756, Aug. 1985.
[8] L. Lamport and P.M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *J. ACM*, vol. 32, no. 1, pp. 52–78, Jan. 1985.
[9] J. Lundelius-Welch and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," *Inform. Computation*, vol. 77, pp. 1–36, 1988.
[10] A. Olson and K.G. Shin, "Probabilistic clock synchronization in large distributed systems," in *Proc. 11th Int. Conf. Distrib. Computing Syst.*, 1991, pp. 290–297.
[11] P. Ramanathan, D.D. Kandlur, and K.G. Shin, "Hardware-assisted software clock synchronization for homogeneous distributed systems," *IEEE Trans. Comput.*, vol. 39, pp. 514–524, Apr. 1990.
[12] S. Rangarajan and S.K. Tripathi, "Efficient synchronization of clocks in a distributed system," in *Proc. Real-Time Syst. Symp.*, 1991, pp. 22–31.
[13] K.G. Shin and P. Ramanathan, "Clock synchronization of a large multiprocessor system in the presence of malicious faults," *IEEE Trans. Comput.*, vol. C-36, no. 1, pp. 2–12, Jan. 1987.
[14] T.K. Srikanth and S. Toueg, "Optimal clock synchronization," *J. ACM*, vol. 34, no. 3, pp. 626–645, July 1987.
[15] N. Vasanthavada and P.N. Marinos, "Synchronization of fault-tolerant clocks in the presence of malicious failures," *IEEE Trans. Comput.*, vol. 37, pp. 440–448, Apr. 1988.
[16] G.M.R. Winkler, "Changes at USNO in global timekeeping," *Proc. IEEE*, vol. 74, no. 1, pp. 151–155, Jan. 1986.



A. Olson received the B.S.E., M.S.E., and Ph.D. degrees in computer engineering from the University of Michigan, Ann Arbor, in 1987, 1989, and 1994, respectively.

He is currently a Research Assistant in the Real-Time Computing Laboratory at the University of Michigan, and is working toward his Ph.D. degree there. His dissertation topic is the synchronization of distributed real-time systems. His primary research interests are in designing, verifying, and testing fault-tolerant and distributed systems.

K. G. Shin (S'75–M'78–SM'83–F'92) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Republic of Korea, in 1970, and the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, USA, in 1976 and 1978, respectively.

He is Professor and Director of the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, USA. From 1978 to 1982, he was on the faculty of Rensselaer Polytechnic Institute, Troy, NY, USA. He has held visiting positions at the U.S. Air Force Flight Dynamics Laboratory, AT&T Bell Laboratories, the Computer Science Division within the Department of Electrical Engineering and Computer Science at the University of California at Berkeley, and the International Computer Science Institute, Berkeley, CA, USA. He also chaired the Computer Science and Engineering Division, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, USA, for three years beginning in 1991. In 1985, he founded the Real-Time Computing Laboratory, where he and his colleagues are currently building a 19-node hexagonal mesh multicomputer, called **HARTS**, to validate various architectures and analytic results in the area of distributed real-time computing. He has also been applying the basic research results of real-time computing to intelligent vehicle highway systems and manufacturing-related applications ranging from the control of robots and machine tools to the development of open architectures for manufacturing equipment and processes. Recently, he has initiated research on the open-architecture information base for machine tool controllers.

Dr. Shin has authored or coauthored more than 270 technical papers (more than 120 of these in archival journals) and numerous book chapters in the areas of distributed real-time computing and control, fault-tolerant computing, computer architecture, robotics and automation, and intelligent manufacturing. He is currently writing a textbook, *Real-Time Systems*, with C. M. Krishna, which is scheduled to be published by McGraw-Hill in 1995. In 1987, he received the Outstanding IEEE TRANSACTIONS ON AUTOMATIC CONTROL Paper Award for a paper on robot trajectory planning. In 1989, he also received the Research Excellence Award from the University of Michigan. He was Program Chair of the 1986 IEEE Real-Time Systems Symposium (RTSS), the General Chair of the 1987 RTSS, the Guest Editor of the 1987 special issue of IEEE TRANSACTIONS ON COMPUTERS (real-time systems), a Program Co-Chair for the 1992 International Conference on Parallel Processing, and served numerous technical program committees. He also chaired the IEEE Technical Committee on Real-Time Systems during 1991–1993, served as a Distinguished Visitor of the IEEE Computer Society, and is an Editor of IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED COMPUTING, and an Area Editor of *International Journal of Time-Critical Computing Systems.*