Efficient Synchronization of Clocks in a Distributed System

Sampath Rangarajan  
Satish K. Tripathi

ABSTRACT Many real-time distributed applications require that the clocks of the processors in a distributed system be synchronized with each other. We propose a probabilistic clock synchronization algorithm where processors in the system exchange time stamps and synchronize to a common clock value. Most of the previous algorithms for this problem have been based on a master-slave approach where all the slave processors synchronize to the clock value of a master. These algorithms are not distributed in nature and some of the assumptions made in these algorithms may become invalid if a large number of slaves try to synchronize with a master. The only distributed algorithm that is known to us [13] is based on finding a cyclic path connecting the processors in the system and exchanging time stamp messages through this path. For the same level of synchronization accuracy, our algorithm uses much smaller number of messages than the algorithm presented in [13].

1 Introduction

We consider the problem of internal clock synchronization where the clocks of processors in a distributed system are kept synchronized relative to each other. As opposed to external clock synchronization where processor clocks are maintained synchronized to an external time reference, internally synchronized clocks may drift away from external time, but stay close to each other. Internal clock synchronization is important in many real-time distributed applications especially in task scheduling where tasks may partly run on one processor and then switched to another processor. In this case, correct measurement of the duration of the task will depend upon how closely the clocks in the two processors are synchronized. Some applications which are executed cooperatively using a group of processors require that the clocks of these processors be synchronized so that deadline constraints and time precedence constraints among the tasks of an application could be satisfied.

Several deterministic [4] [5] [9] [10] [14] and a few probabilistic algorithms [2] [1] [13] have been proposed for clock synchronization. All these algorithms are based on exchanging time stamp messages between the processors in the system. These messages face unpredictable communication delays. Given that \( d_{\text{max}} \) and \( d_{\text{min}} \) are the maximum and minimum communication delay of a time stamp message, the deterministic algorithms have a lower bound [11] [3] on the deviation (after synchronization) between clocks that they can guarantee. For a system with \( N \) clocks, this lower bound is given by \( \gamma \geq (d_{\text{max}} - d_{\text{min}})(1 - \frac{1}{N}) \). These algorithms guarantee a deviation of \( \gamma \) between clocks with certainty. Probabilistic algorithms relax this restriction by guaranteeing a deviation between clocks of \( \gamma \) not with certainty, but with high probability. With these algorithms, \( \gamma \) need not satisfy the lower bounds shown for the deterministic algorithms. In fact, it could be made arbitrarily close to zero.

We are interested in the design of efficient probabilistic clock synchronization algorithms. Very few such algorithms have been proposed so far. Cristian [a] proposed probably the first such algorithm where a processor wishing to synchronize sends a messages to a master asking for its time stamp. If a reply from the master does not arrive within a specified time-out period, it retransmits the message until it receives a time stamp message from the master within the time-out period. It was shown that this algorithm can achieve, with very high probability, a deviation between clocks much smaller than the lower bound on \( \gamma \) for the deterministic algorithms. The algorithm presented in [1] is based on averaging out the delays of time stamp messages. The algorithm uses multiple time stamp messages and through the process of averaging filters out the variation in the delay of these messages. Like the algorithm presented in [2], this algorithm achieves clock deviations much smaller than the deterministic lower bound with a probability of such a guarantee not holding decreasing exponentially towards zero with an increase in the number of time stamp messages. The above two algorithms are based on a master-slave approach where all processors in the system synchronize to the clock of a master. The slaves read the clock time of the master by sending synchronization messages and receiving the time stamp from the master. Multiple
such messages are used to filter out the error in the readings due to communication delay by the process of averaging. When all the processors in a large system are to be synchronized, this will lead to "hot spots" in message transmission where a master gets flooded with messages from all the slaves. Also, different slaves will be at different communication distances from the master and some of the assumptions made in the above algorithms about communication delays may not hold.

In [13], a probabilistic algorithm is presented which does not use a master-slave approach. In this algorithm, multiple time stamp messages are exchanged among a group of processors using a cyclic path and these time stamp messages are used by every processor in the group to estimate the value of every other processor by using either an averaging approach or an interval oriented approach [2]. A processor needs to estimate the clock values of only the processors with which it is grouped, and not every other processor in the system. Processors common to groups are used as "bridges" between the groups for synchronization. If processors within a group are synchronized to within $\gamma$, then two processors in different groups bridged together by a common processor will be within $2\gamma$. It is shown that the accuracy of clock estimation decreases with the increase in the size of the group. But smaller sized groups will lead to many such groups and hence potentially large deviation in clocks (after synchronization) between two clocks which belong to two different groups which are far apart.

In this paper, we propose a solution for synchronizing all the clocks of a distributed system with much fewer messages than the solution presented in [13]. Our solution is based on a very efficient scheme for exchanging time stamps between processors in the system. In Section 2 we develop a framework for measuring the efficiency of probabilistic clock synchronization algorithms. In particular, we give precise meaning to the term "high probability" which enables us to compare the asymptotic overhead of different probabilistic algorithms. In Section 3 we discuss techniques for efficient message exchange. We present our algorithm in Section 4 and analyze it in Section 5. Section 6 discusses the use of our algorithm for synchronizing large systems and presents numerical results. Conclusions are made in Section 7.

## 2 Overhead for Clock Synchronization Algorithms

Overhead for deterministic clock synchronization algorithms are measured in terms of the total number of messages that are required to be exchanged among all the processors in the system to reach synchrony (or near synchrony). We need an equivalent figure of merit for probabilistic algorithms. So, let us first look at a common framework for measuring the overhead of such algorithms. As a common denominator for measuring message complexity, we assume that data sent from one processor to its neighbor (one hop) constitutes a message.

### 2.1 Overhead for probabilistic clock synchronization algorithms

All these algorithms are based on processors estimating the clock values of one another. Assume that there are totally $N$ processors in the system. We will consider first, the probability that the estimation of the clock value $T_i$ of a processor $i$ by another processor $j$ is different from $T_j$ by more than $\epsilon_{max}$, where $\epsilon_{max}$ is the maximum error that we could allow in these estimations (in order to keep the clocks synchronized within $\gamma$). Let this probability $P(\epsilon > \epsilon_{max})$ be denoted by $P_{error}$. This is also referred to as the probability of invalidity. Depending on the algorithm used, $\epsilon_{max}$ is determined by the maximum deviation $\gamma$ that we are willing to allow between any two clocks in the system (or a group for the algorithm in [13]) after synchronization. We could guarantee such a deviation only if each of the processors in the system read the clocks of every other processor (or every other processor in its own group for the algorithm in [13]) in the system with an error not exceeding $\epsilon_{max}$. Let $E$ be the event that this happens. Then, what we would like to measure is the number of time stamp messages that have to be transmitted in the system for the whole synchronization process so that $P(E)$ does not decrease (stays constant or increases towards 1) when the system size increases (i.e. $N \rightarrow \infty$). We would refer to an algorithm where this condition holds as an algorithm which works with asymptotically high probability.

Given that each processor estimates the clock value of every other processor, there are $N \times N - 1 \approx N^2$ such estimates that are made. If $P_{error}$ is independent for each such estimate, then $P(E) = (1 - P_{error})^N$ and it can be shown that $P(E) \geq 1 - N^2P_{error}$. But the error made during each estimate may not be independent. For example, for the
algorithm in [13], because of the way the messages are organized and sent, a processor reads the clock values of all other \( N - 1 \) processors from a single message with a certain error. So, once we calculate \( P_{\text{error}} \) for different algorithms, it will be possible to calculate \( P(E) \) and compute the overhead in terms of the number of messages for those algorithms to work with asymptotically high probability.

### 2.1.1 Overhead for master-slave approaches

Let us look at the algorithms presented in [2] and [1] and evaluate their overhead under our framework. For the algorithm presented in [2], for a certain precision \( \epsilon \) of estimation of the master clock, \( P_{\text{error}} \) goes to zero exponentially with an increase in the number of clock readings (of the master) per slave processor. Now, each slave processor estimates the clock time of only the master and with \( N \) such slaves, there are only \( N \) such (independent) estimations. So, \( P(E) \geq 1 - N P_{\text{error}} \). Given that \( P_{\text{error}} \) goes to zero exponentially with the number of clock readings, this gives,

\[
P(E) \geq 1 - N c_1 \exp(-c_2 r)
\]

where \( r \) is the number of clock readings per slave processor and \( c_1 \) and \( c_2 \) are constants. It is easy to see that \( r = O(\log N) \) for asymptotically high probability of \( P(E) \) (ie. \( P(E) \) does not decrease when \( N \) increases). Every time the master clock is read by a slave, two messages are sent, one in each direction. For a total of \( N \) slaves, this leads to a message complexity of \( O(N \log N) \). Similarly, for the algorithm presented in [1], it is shown that \( P_{\text{error}} = \alpha \exp(-r\beta) \) where \( r \) is again, the number of clock readings (of the master) per slave processor, \( \alpha = O(\frac{1}{\sqrt{r}}) \) and \( \beta \) is a constant. Then with \( N \) slaves,

\[
P(E) \geq 1 - N \frac{1}{\sqrt{r}} c_1 \exp(-r\beta)
\]

where \( c_1 \) is a constant. Again, it can be shown that for asymptotically high probability of \( P(E) \), \( r = O(\log N) \) for a total message complexity of \( O(N \log N) \).

For the two algorithms above, we have assumed that time stamp messages from the master reaches each of the slaves in one hop. This is not normally the case and so the real message complexity will be greater than what is shown above.

### 2.1.2 Overhead for the distributed approach

The algorithm presented in [13] does not follow a master-slave approach, but instead is based on every processor reading every other processor’s clock in a group of processors. To make all the processor clocks in a system of \( N \) processors to be within \( \gamma \) of each other after synchronization, all processors have to be grouped together and time stamp messages have to be sent through a hamiltonian cycle connecting all processors. If every processor’s clock value is sent \( r \) times to every other processor, it is shown that

\[
P_{\text{error}} = 1 - \text{Erf} \left( \frac{\text{max}\sqrt{r}}{\sqrt{N} \sigma} \right)
\]

where \( \sigma^2 \) is the variance of message delay due to a message going from one node (processor) to another and \( \text{Erf} \) is the error function for the normal distribution. Substituting for \( \text{Erf} \), we get

\[
P_{\text{error}} = \sqrt{\frac{2}{\pi}} \frac{\sigma}{\sqrt{\text{max}} \sqrt{r}} \exp \left( -\frac{\text{max} r}{N \sigma^2} \right)
\]

Now, although the clock value of every processor is read by every other processor (there are \( N^2 \) readings), because the way messages are sent, a processor reads the clock values of all processors from a single message. There will be only \( N \) such readings and so, \( P(E) \geq 1 - N P_{\text{error}} \). For asymptotically high probability of \( P(E) \), it is easy to see that \( r = O(N \log N) \). Each time a processor clock is read, the message containing the time stamp goes through all the \( N \) processors (ie. it takes \( N \) hops) and for every hop, a new time stamp (for the processor in that hop) is added and sent to the next processor down the line. Thus, the total message complexity is \( O(N^2 \log N) \). This algorithm has a message complexity much higher than the master-slave approaches because every clock value is read by every other clock. Also, in the master-slave approaches, we ignored the many hops that a message might potentially take to reach a slave from the master.

### 3 Efficient exchange of time stamp messages

In the algorithm presented in [13], a cyclic path is used to transmit time stamp messages. It is shown that the accuracy of estimation becomes worse with large number of processors to a group. When a cyclic path is used to send time stamp messages between processors in a group, the number of hops a message takes on an average to reach its destination grows linearly with the number of processors. The more the number of hops the time stamp message takes to reach its destination, the larger the delay it incurs and less accurate the clock reading. Further,
Figure 1: Grouping processors with \( \sqrt{N} \) processors per group

in [13], it does not seem obvious how to set up the groups of processors. For example, it is shown how in a hypercube multiprocessor, a particular choice of groups leads to the possibility of not being able to keep the system synchronized at all. As was shown previously, if the whole system is to be synchronized by having all the processors in a single group, the message complexity is very high.

Given that each processor has to read every other processor's clock, and given that the accuracy of clock readings decrease with more delay per time stamp message, the problem can be narrowed down to the efficient exchange of time stamps with as small a delay as possible. That means, we should make use of the connectivity of the network more efficiently to send time stamp messages.

3.1 Grouping processors for efficient message exchange

Let us motivate our approach with a specific communication strategy. In [12], it is shown how \( N \) processors in a distributed system can be divided into groups with \( \sqrt{N} \) processors per group, such that any two groups have a non-null intersection. If \( S_i \) and \( S_j \) are two groups of \( \sqrt{N} \) processors each, then \( S_i \cup S_j \neq \emptyset \). There are \( N \) such groups, where each group is associated with a particular processor. It can be shown then that each processor belongs to \( \sqrt{N} \) groups. The situation, where \( N \) is not a square leads to some dummy processors being added. The groups are formed using finite projective planes [12]. An example is shown in Figure 1 (from [12]) where there are 7 nodes, and 3 groups with 3 processors per group (the number closest to 7 which is a square is 9). Also, each processor belongs to 3 groups and any two groups have a non-null intersection. \( S_i \) refers to the group associated with processor \( i \). Using this grouping technique, efficient distributed consensus protocols were developed in [6]. It can be shown that if all processors in a group are directly connected to one another, then a message (in our case, a time stamp message) from all processors could reach all other processors in two hops (steps). The communication protocol would work as follows. In the first step, all processors \( i, 1 \leq i \leq N \), would send their own time stamp to all other processors in their associated set \( S_i \) and all processors \( j \) such that \( i \in S_j \). In the second step, the processors will repeat the first step with a message now containing all the time stamps that each of them received in the first step. Because every pair of groups have a non-null intersection, every processors' time stamp reaches every other processor in two hops (steps). For example, in Figure 1, in every step processor 1 would send time stamp messages to processors 2 and 3 in its associated set and processors 4 and 6 because 1 belongs to \( S_4 \) and \( S_6 \). It can be seen that in each of the two steps, each of the \( N \) processors send messages to \( 2(\sqrt{N} - 1) \) other processors for a total message complexity (for the two steps) of \( O(N\sqrt{N}) \). Compare this to a one step communication protocol where (if every processor is directly connected to every other processor), the message complexity would be \( O(N^2) \).

3.2 A more efficient grouping technique

The above protocol takes only two steps, but then every group is of size \( \sqrt{N} \). If a processor has to send a message to every member of its associated group in one step, then every processor has to be directly connected to at least \( O(\sqrt{N}) \) other processors. In [7], a grouping strategy is proposed which extends the above idea. In this strategy, groups are formed in such a way that there are \( k(N^{\frac{1}{k}} - 1) + 1 \) processors to a group and every processor belongs to \( k(N^{\frac{1}{k}} - 1) + 1 \) groups. There are \( N \) such groups each associated with a particular processor. Note that \( k = 2 \) would lead to the \( \sqrt{N} \) grouping strategy discussed above (up to a constant). If \( N^{\frac{1}{k}} \) is not an integer, then some dummy nodes will be added. With this grouping strategy, unlike in the previous case, a pair of groups may have a null intersection, but there is a path of length at most \( k \) connecting any two processors in different groups. That is, by increasing the number of steps for a message to reach every other processor in the system, the group size (or the connectivity requirement) has been decreased. The groups can be formed using \( k \) dimensional hypercubes [7]. A processor \( i \) would be numbered by \( (i_1, i_2, ..., i_k) \), where \( 1 \leq i_1, i_2, ..., i_k \leq N^{\frac{1}{k}} \). A set of \( N \) groups is constructed by associating with each of the \( N \) processors, a group consisting of itself and all other processors whose number differ from \( i \) by \( \leq k \).
one of the processor under consideration) in exactly one of the \( k \) elements of the \( k \) tuple. For a processor \( i = (i_1, i_2, \ldots, i_k) \), its subset is \( (i_1, i_2, \ldots, i_k) \cup (*, i_1, i_2, \ldots, i_k) \cup (i_1, *, i_3, \ldots, i_k) \cup \ldots \cup (i_1, i_2, i_3, \ldots, *) \). The * in the \( j \)th position indicates that the element in that position can take any value between 1 and \( N^\frac{1}{k} \), excluding \( i_j \). The communication protocol using this grouping structure is as follows. In the first step, all processors \( i, 1 \leq i \leq N \), would send a message with their own time stamp to all other processors in their associated set \( S_i \) and all processors \( j \) such that \( i \in S_j \). In the subsequent steps, each of the \( S_i \) will repeat the first step, except the message that they send out will contain all the time stamps from the messages that they received in the previous step. Note that, time stamp of a processor may reach another processor multiple number of times through different messages, but because any two processors are at a distance at most \( k \), within \( k \) steps, every processor will receive every other processors’ time stamp. In each of the \( k \) steps, each of the \( N \) processors send messages to \( 2k(N^\frac{1}{k} - 1) \) other processors for a total message complexity of \( O(k^2N \times N^\frac{1}{k}) \). Note that for \( k = 2 \), this is \( O(N\sqrt{N}) \). The value of \( k \) could be a constant or it could be a function of \( N \). It can be shown that the message complexity is minimum when \( k = \log_b N \) where \( b \) is some base. Then, \( O(k^2N \times N^\frac{1}{k}) \) is equal to \( O(N \log^2 N) \).

In the next section, we will show how a protocol for time stamp message exchange using the above grouping strategy could be used to design a very efficient clock synchronization algorithm.

4 The Algorithm

Let us consider a set of \( N \) processors which can be grouped according to the grouping strategy presented in the last section. Then, every processor can reach \( k(N^\frac{1}{k} - 1) \) other processors in one hop (step). Let \( S_i \) be the group of processors associated with processor \( i \). The algorithm works in two stages. In the first stage, processors send their time stamps to each other. In the second stage, a processor uses the time stamps it received to estimate the time stamps of all the other processors and then synchronize to the average of those estimations.

4.1 First Stage

In the first stage, each processor \( i \) sends its time stamp \( r \) times to every other processor as follows.

1. In step \( s, 1 \leq s \leq r \) send a message to all processors in \( S_i \) and all processors \( j \) such that \( i \in S_j \). The message contains the \( s_{th} \) time stamp of \( i \) and all the time stamps that were received by \( i \) in the \( s - 1_{th} \) step.

2. In step \( s, r < s \leq m \), send a message to all processors in \( S_i \) and all processors \( j \) such that \( i \in S_j \). The message now contains only all the time stamps that were received by \( i \) in the \( s - 1_{th} \) step.

In the above algorithm for exchanging time stamps, the time stamp messages are pipelined. That means, through the messages that a processor receives in the \( s + 1_{th} \) step, it will get the \( s_{th} \) time stamp of the processors which are at a distance 1 away, \( s - 1_{th} \) time stamp of the processors which are at a distance 2 away and so on.

During step \( s, 1 \leq s \leq r \), processors send out new time stamp messages. In step \( s, r < s \leq m \), processors clear the remaining messages carrying the time stamps that are still in transit to their destinations. Given that any processor is within distance \( k \) from any other processor, from the time the last time stamp is generated by a processor and sent out as part of a message, there will be an additional \( k - 1 \) steps before these time stamps reach their destinations. So, there will be totally \( m = r + (k - 1) \) steps in the algorithm. Note that all the processors need not start a step at the same time. But a processor \( i \) starts its \( s_{th} \) step only after all the messages sent in the (respective) \( (s - 1)_{th} \) step by other processors in the groups to which \( i \) belongs, have arrived. So, after \( m = r + (k - 1) \) steps, every processor would have the \( r \) time stamps of every other processor. As we mentioned in the previous section, a time stamp of a processor may arrive at another processor multiple number of times as part of different messages. All but the first such arrival can be ignored. It may be possible to increase the fault tolerance of the algorithm by considering subsequent arrivals, but we have not explored that possibility in detail.

At every step, \( N \) processors send messages to \( (\text{maximum}) \ 2k(N^\frac{1}{k} - 1) \) other processors. There are a total of \( m \) steps. So, the total message complexity would be \( m \times N \times 2k(N^\frac{1}{k} - 1) \).

4.2 Second Stage

In the second stage, each processor estimates the clock time of other processors as follows. Let \( T_{ij}^s \) be the \( s_{th} \) time stamp received by processor \( i \) from processor \( j \). Let the time shown by \( i \)'s local clock
when \(T_{i,j}^t\) arrives be \(LT_{i,j}^t\). Assume that the first arrival of the first time stamp from \(j\) at \(i\) is at the end of the \(s_{th}\) step. This means that \(i\) is at distance \(s\) from \(j\). Note, that because time stamp messages are pipelined, subsequent time stamps will arrive one per step. Let \(ET_{i,j}\) be the estimate of the \(j\)'s clock by \(i\) \((ET_{i,j}^t\) will be the reading of processor \(i\) of its own clock value). This estimate is computed by processor \(i\) using an averaging formula similar to the one in \([1]\) and \([13]\).

\[
ET_{i,j} = LT_{i,j}^t - \left(\frac{1}{r} \sum_{t=1}^{r} (LT_{i,j}^t - T_{i,j}^t - s\mu)\right)
\]

Here, \(\mu\) is the expected delay for a message per hop (step). Once, processor \(i\) estimates the clock values of all other \(N - 1\) processors, it readjusts its time to the average of all the clock values (including its own clock), given by the equation

\[
\frac{1}{N} \sum_{j=1}^{N} ET_{i,j}
\]

### 5 Asymptotic Analysis

In this section, let us analyze the overhead of our algorithm for asymptotically high probability of synchronization (ie. \(P(E)\) stays a constant when \(N \to \infty\)).

First, we need to compute the probability \(P(\epsilon > \epsilon_{max})\). This is the probability \(P_{error}\) that the error in a single clock reading \(\epsilon\) is greater than \(\epsilon_{max}\). Let \(AT_j\) be the actual time of \(j\) when the \(r_{th}\) time stamp arrives at \(i\). \(ET_{i,j}\) is the estimate of \(j\)'s time by \(i\) and so the error in the clock reading \(\epsilon\) is \(ET_{i,j} - AT_j\). Now the actual time at \(j\) will be the sum of the time shown on the \(r_{th}\) time stamp which is \(LT_{i,j}^t \) and the total delay (on all \(s\) hops) that the time stamp incurred in reaching \(j\) from \(j\). That is, \(AT_j = LT_{i,j}^t + d_s\). Then, substituting for \(ET_{i,j}\) from equation above,

\[
\epsilon = ET_{i,j} - AT_j = \frac{1}{r} \sum_{t=1}^{r} (T_{i,j}^t - LT_{i,j}^t) + s\mu + d_s
\]

The equation for \(\epsilon\) is a sum of independent random variables. So, when the number of time stamps \(r\) is large, we can approximate it to a normal distribution using central limit theorem. The mean of each of the terms in the sum is \(\bar{\epsilon}\) which is the mean of the error \(\epsilon\) and if \(\sigma^2\) is the variance of the delay (per hop), then it can be shown using analysis similar to the one found in \([1]\) that the probability distribution of \(\epsilon\) approaches the normal distribution \(N(\bar{\epsilon}, \frac{\sigma^2}{r})\) when the number of time stamps \(r\) increases. From the above approximation,

\[
P(\epsilon \leq \epsilon_{max}) = \text{Erf}\left(\frac{\epsilon_{max}\sqrt{r}}{\sqrt{2}\sigma}\right)
\]

where \(\text{Erf}\) is the error function for normal distribution. Then,

\[
P_{error} = P(\epsilon > \epsilon_{max}) = 1 - \text{Erf}\left(\frac{\epsilon_{max}\sqrt{r}}{\sqrt{2}\sigma}\right)
\]

It can be shown \([1]\) that

\[
P_{error} = \sqrt{\frac{2}{\pi}} \frac{\sigma}{\sqrt{\epsilon_{max}}} \exp\left(-\frac{\epsilon_{max}^2 r}{2\sigma^2}\right)
\]

This is the error made in one clock reading of processor \(j\) by processor \(i\). The time stamp from \(j\) reaches \(i\) in \(s\) steps. But in the worst case, two processors are at a distance \(k\) apart and the time stamp will take \(k\) steps to reach the destination. So, we can get the worst case value for \(P_{error}\) by substituting \(k\) for \(s\) in the above equation. There are \(N^2\) such clock readings (every processor reads every other clock).

As discussed in Section 2, \(P(E) \geq 1 - N^2 P_{error}\) if all the \(N^2\) clock readings are independent. In our algorithm, the \(N^2\) clock readings are not independent, but for upper bound calculations, we will consider the worst case and assume that it is true (if there were only \(N\) independent clock readings and not \(N^2\) such readings, it will affect only the constants). Then, the following theorem gives a bound on \(r\), the number of time stamps that is sufficient to be sent from one processor to another for asymptotically high probability of \(P(E)\).

**Theorem 1** With \(r \geq \frac{2\sigma^2}{\epsilon_{max}} k \log_e N\), we can achieve asymptotically high probability of \(P(E)\). That is \(P(E)\) stays a constant or increases towards 1 (in this case, it increases towards 1) when \(N \to \infty\).

**Proof:** From the above discussion,

\[
P(E) \geq 1 - N^2 \sqrt{\frac{2}{\pi}} \frac{\sigma}{\sqrt{\epsilon_{max}}} \exp\left(-\frac{\epsilon_{max}^2 r}{2\sigma^2}\right)
\]

Let \(r = c_1 \log_e N\) where \(c_1\) is a constant. Substituting this in the above equation, we get

\[
P(E) \geq 1 - \frac{\sigma}{\epsilon_{max}} \sqrt{\frac{2}{\pi}} \frac{\sqrt{\epsilon_{max}}}{{c_1 \log_e N}} N(2 - \frac{2\epsilon_{max}^2 c_1}{k^2})
\]
If \( c_1 = \frac{2k\sigma^2}{\epsilon_{\text{max}}} \), then the above equation becomes
\[
P(E) \geq 1 - \frac{1}{k\pi \log_2 N}
\]
When \( N \to \infty \), it can be seen that \( P(E) \) approaches 1. So, with \( r = \frac{2\sigma^2}{\epsilon_{\text{max}}} k \log_2 N \), we can achieve asymptotically high probability of \( P(E) \).

Let us now compute the message complexity of our algorithm. It can be shown that the message complexity is minimum when \( k = \log_b N \) (where \( b \) is some base). The following lemma gives that message complexity.

**Lemma 1** For \( k = \log_b N \), the total message complexity for the clock synchronization algorithm is \( O(N \log^2 N) \).

**Proof:** From the previous section, the total message complexity (denoted by \( M \)) is \( M = m \cdot N \cdot 2k \cdot (N^{\frac{1}{2}} - 1) \). The number of steps \( m \) is \( r + (k - 1) \). So, \( M = (r + k - 1) \cdot N \cdot 2k \cdot (N^{\frac{1}{2}} - 1) \). Also, from Theorem 1, \( r = \frac{2\sigma^2}{\epsilon_{\text{max}}} k \log_2 N \). Substituting this values in the equation for \( M \), we get,
\[
M = \left[ \left( \frac{2\sigma^2}{\epsilon_{\text{max}}} \right) k \log_e N + k - 1 \right] \cdot N \cdot 2k \cdot (N^{\frac{1}{2}} - 1)
\]
It can be shown that this value is a minimum when \( k = \log_b N \). Then, we get
\[
M = 2 \left[ \left( \frac{2\sigma^2}{\epsilon_{\text{max}}} \right) \log_b N \log_e N + \log_b N - 1 \right] \cdot N(b - 1) \log_b N
\]
from which \( M = O(N \log^2 N) \).

To get the above message complexity we need \( k = \log_b N \). We know that the size of a group is \( k(N^{\frac{1}{2}} - 1) + 1 \). For \( k = \log_b N \), this is \( O(\log N) \). That means a processor should be able to communicate with \( O(\log N) \) other processors in one hop (step). For \( b = 2, k = \log_2 N \) and the group size is exactly \( \log_2 N \). So, in this case, a hypercube interconnection structure would satisfy the requirement. Even if such \( O(\log N) \) connectivity is not available, in the next section, we show how a grouping technique such as the one we used above could be used together with an algorithm like the one in [13] (where processors in a group are connected using a cyclic path) to design an algorithm whose efficiency is only \( O(\log N) \) off the result presented in Lemma 1.

### 6 Issues in Large System Synchronization

#### 6.1 Synchronizing a large system

When a large system is to be synchronized using the algorithm presented above, we need to compute the maximum error in clock estimation \( \epsilon_{\text{max}} \) that we can allow, so that after synchronization all clocks will be within a specified deviation. Let us consider a system, where all the clocks are synchronized to within \( \delta \) of each other. This is the initial state of the system before synchronization. We would like to run our synchronization algorithm and at the end of the algorithm, assume that we would like to have the clocks in the system to be within \( \gamma \) of each other, where \( \gamma < \delta \). \( \gamma \) is a function of \( \epsilon \), which is the estimation error that is made when one processor estimates the clock value of another processor. The following theorem relates \( \gamma \) to \( \epsilon \).

**Theorem 2** In a system with \( N \) clocks, for clocks to be within \( \gamma \) of each other after synchronization, we need to ensure that \( \epsilon \leq \frac{N\gamma}{2(N-1)} \).

**Proof:** Consider two processors \( a \) and \( b \) with clocks \( A \) and \( B \) respectively which are synchronized to within \( \delta \). Given the initial state of the system, all other \( N - 2 \) clocks are also synchronized to within \( \delta \) of both \( A \) and \( B \). Without loss of generality, assume that \( A \) shows the smallest clock reading in the system. Let it be denoted by \( T_A \). Now, the clock value of \( B \) would be \( T_A + \delta \). Processors \( a \) and \( b \) get all clock values in the system (including their own clock values) and compute the average and reset clocks \( A \) and \( B \) respectively to that time. Let the clock value of processor \( i, i \neq a, b \) be \( T_A + t_i \), for \( 1 \leq i \leq N - 2 \) where \( 0 \leq t_i \leq \delta \) (because, \( A \) is assumed to show the smallest clock value). The maximum value of clock \( A \) after synchronization would be (see Figure 2)

\[
T_{\text{new}A} = T_A + T_A + \delta + \epsilon + T_A + t_1 + \epsilon + \ldots + T_A + t_{N-2} + \epsilon\]

and the minimum value of clock \( B \) after synchronization would be

\[
T_{\text{new}B} = T_A + \delta + T_A - \epsilon + T_A + t_1 - \epsilon + \ldots + T_A + t_{N-2} - \epsilon\]

The maximum difference in the two clock values would be\( T_{\text{new}A} - T_{\text{new}B} \) which is \( \frac{2(N-1)\epsilon}{N} \). The maximum allowable value of this difference is \( \gamma \). Then \( \gamma \geq \frac{2(N-1)\epsilon}{N} \), from which \( \epsilon \leq \frac{N\gamma}{2(N-1)} \). \( \square \)
The above result tells us that the maximum value of error in a single clock estimation ($\epsilon_{\text{max}}$) that we can allow is $\epsilon_{\text{max}} = \frac{N\gamma}{2(N-1)}$. For large values of $N$, we see that $\epsilon_{\text{max}} \approx \frac{\gamma}{2}$. Substituting for $\epsilon_{\text{max}}$ in the equation for $r$ in Theorem 1, we can state the following corollary.

**Corollary 1** If the number of time stamps ($r$) sent from one processor to another is $r \geq 8 \left( \frac{(N-1)\gamma}{N^2} \right)^2 \log_e N \geq 8 \left( \frac{\gamma}{2} \right)^2 \log_e N$, then after synchronization, every clock is synchronized to within $\gamma$ of every other clock with asymptotically high probability.

\[ \Box \]

Another parameter of interest is the time between synchronizations. Assume that we allow the clocks to drift away from each other up to a maximum deviation of $\delta$. After every synchronization, the clocks are readjusted to be within $\gamma$ ($\gamma < \delta$). Let $\rho$ be the maximum drift rate of the clocks. Then, the time between synchronizations is $\tau_{\text{sync}} \leq \frac{\delta - 2}{2\rho}$. Similar observation is made in [13].

### 6.2 Evaluation of the message overhead

Numerical evaluation of our results for specific values of different parameters will give us an idea about the message overhead of these algorithms in practice. Let us consider a binary hypercube interconnection structure. Then $k = \log_2 N$ (i.e., base $b = 2$).

Consider the proof of Lemma 1. The message complexity is given by (with $b = 2$),

\[
M = 2 \left[ \left( \frac{2\sigma^2}{\epsilon_{\text{max}}^2} \right) \log_2 N \log_e N + \log_2 N - 1 \right] + N \log_2 N
\]

The above message complexity was derived using the fact that every processor $i$ sends messages to at most $2k(N^k - 1)$ other processors in a step ($k(N^k - 1)$ processors in $S_i$ and $k(N^k - 1)$ other processors whose associated sets contain $i$). Set $S_i$ may contain some nodes $j$, where $i \in S_j$. For $k = \log_2 N$, it can be shown that $S_i$ contains all processors $j$, such that $i \in S_j$. So, every processor will send messages only to $k(N^k - 1) = \log_2 N$ processors in a step (This shows that $\log_2 N$ connectivity is sufficient). Then, the message complexity becomes,

\[
M = \left[ \left( \frac{2\sigma^2}{\epsilon_{\text{max}}^2} \right) \log_2 N \log_e N + \log_2 N - 1 \right] + N \log_2 N
\]

Assume that we need the clocks to be synchronized within $\gamma$ of each other. Then, using the result from Theorem 1 and substituting $\epsilon_{\text{max}} = \frac{N\gamma}{2(N-1)}$ in the above equation, we get

\[
M = \left[ 8 \left( \frac{(N-1)\gamma}{N\gamma} \right)^2 \log_2 N \log_e N + \log_2 N - 1 \right] + N \log_2 N
\]
That is, $M$ is no greater than
\[
\left[ 8 \left( \frac{\sigma}{\gamma} \right)^2 \log_2 N \log_e N + \log_2 N - 1 \right] \cdot N \log_2 N
\] (1)
The number of time stamp messages required in this case will be (from Corollary 1, with $k = \log_2 N$)
\[
r \geq 8 \left( \frac{(N-1)\sigma}{N\gamma} \right)^2 \log_2 N \log_e N
\]
That is,
\[
r = 8 \left( \frac{\sigma}{\gamma} \right)^2 \log_2 N \log_e N
\] (2)

Also, from Theorem 1, the probability of synchronization $P(E)$ is
\[
P(E) \geq 1 - \frac{1}{\pi \log_2 N \log_e N}
\] (3)

Table 1 shows the number of time stamps $r$ that are sufficient (to be sent from a processor to other processors) for asymptotically high probability of correct synchronization. The standard deviation of the delay of a message (per hop) is assumed to be $\sigma = 1.0$ milliseconds. We have tabulated (using Equation 2) $r$ for $\gamma$ (maximum deviation between clocks after synchronization) values of 2, 4, and 6 milliseconds. Similar numbers were used in [1]. It is to be noted that these numbers are based on the upper bounds that we derived and actually are overestimations. We consider hypercubes of size 16, 32 and 64. It is not straightforward to compare these numbers with the ones in [13]. The number of time stamps computed in [13] are for a $P_{error}$ of $\epsilon$ (see the discussion in Section 2.1.2) and not for $P(E)$. So, the actual number of time stamps will be more than what is shown. Also, we compute $r$, not for a specific value of $\epsilon$, but for a specific value of $\gamma$. In [13] the relationship between $\gamma$ and $\epsilon$ is dependent on how the processors are grouped.

Let us take a specific example in [13] and try to compare it to our numbers. A 32 processors set up is presented where processors are grouped in such a way that every processor is within a (group) distance of two of every other processor. Assume that we want all processor clocks to be within 8 ms of each other after synchronization. Then, all processor clocks in a group have to be within 4 ms of each other. That is, $\gamma_{group}$ within a group has to be 4 ms. For this set up, it is shown in [13] that $\gamma_{group} > \frac{15}{4} \epsilon$. So, $\epsilon$ is $\approx 1.1$ ms. For this value of $\epsilon$ (and for $\sigma = 1$ ms), the number of time stamps is shown to be 22. From Table 1, we see that it is 3 for our algorithm (see the tabulation for $\gamma = 8$). Further, number of time stamps increases at least linearly with the group size for the algorithm in [13], whereas ours increases polylogarithmically ($O(\log^2 N)$). We should also point out that we have measured $r$ at the point $P(E)$ starts increasing asymptotically (towards 1) with an increase in $N$. For $N = 32$, from Equation 3, we can compute $P(E)$ at this point to be $\geq 0.86$. The value for the number of time stamps calculated in [13] is for $P(\epsilon < \epsilon_{max}) = 0.9$ (that is, $P(\epsilon \geq \epsilon_{max}) = P_{error} = 0.9$) and $P(E)$ in this case should be much smaller than 0.9.

We can use Equation 1 to compute the total number of messages. For the above example, it turns out to be 64 messages per processor. Note that we count data that is sent from one processor to its neighbor in a step (in one hop) as a message. The number of messages required by the algorithm in [13] will be much more. To reiterate, if all processors are connected in a cycle (in [13]), the message complexity for our algorithm is no greater than $O(N \log^3 N)$.

### 6.3 A modified algorithm for systems with less connectivity

Let us consider the situation where a connectivity of $O(\log N)$ is not available. We can still group the processors using the same technique as before with $k = \log_2 N$. That is, group the processors, such that there are $k(N^{\frac{1}{k}} - 1) + 1 = O(\log N)$ processors to a group. There will be $N$ such groups. We can use a technique similar to the one in [13] for synchronization of processors within a group. That is, form a cycle connecting all the processors in the group and exchange time stamp messages through this path. There are $N$ groups, however any two processors in the system are at most $k = \log_2 N$ groups away from each other. Then, if we want any two processors in the system to be within $\gamma$ of each other after synchronization, we need to have all the processors in a group synchronized to within $\frac{2}{\log_2 N}$ of each other. From Theorem 2, this means that the maximum clock reading error that we could allow for processors will be $\frac{2\epsilon_{max}}{\log_2 N}$.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\gamma = 2$ ms</th>
<th>$\gamma = 4$ ms</th>
<th>$\gamma = 6$ ms</th>
<th>$\gamma = 8$ ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>23</td>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>35</td>
<td>9</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>64</td>
<td>50</td>
<td>13</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1: Number of time stamps ($r$) for different system sizes
where \(c_1\) is some constant. This just tells us that when the system size increases, the error that we can allow in a clock reading has to go down as \(O(\frac{1}{\log N})\).

Assume that we run the algorithm of [13] within a group of \(O(\log N)\) processors using a cyclic path (A hamiltonian cycle can always be found if the communication links are bidirectional). Every node will have an even degree. When the system size increases, the error that we can allow in a clock reading has to go down as \(O(\frac{1}{\log N})\).

Then, from the discussion in Section 2.1.2, we can see that for such an algorithm,

\[
P_{error} = c_1 \sqrt{\frac{2}{\pi}} \frac{\sigma \log N}{r} \exp \left( \frac{-c_2 r^2}{\sigma^2 \log^3 N} \right)
\]

where \(c_1\) and \(c_2\) are constants. Because the group size is \(O(\log N)\), \(P(E) \geq 1 - O(\log N) \cdot P_{error}\). Substituting for \(P_{error}\) from above, it can be shown that \(r = O(\log^3 N)\) for asymptotically high probability of \(P(E)\). Each of these \(r\) time stamps go through \(O(\log N)\) processors (hops), for a total message complexity inside a group of \(O(\log^4 N)\). There are totally \(N\) such groups and so the total message complexity for the whole system would be \(O(N \log^4 N)\). Thus, even if we do not have \(O(\log N)\) connectivity, simply by using the grouping scheme, we can design an algorithm which is only \(O(\log N)\) off the message complexity of the algorithm presented in Section 4 (Lemma 1 gives its message complexity).

### 7 Conclusions

It is very important in many real-time distributed applications to keep the clocks of the processors in the system synchronized with each other. We presented a probabilistic algorithm for distributed clock synchronization, which achieves asymptotically high probability of synchronization with an overhead of at most \(O(N \log^3 N)\) messages. Time stamp messages are exchanged very efficiently through a processor grouping scheme where every processor exchanges messages with \(O(\log N)\) processors in one step. If \(O(\log N)\) connectivity was not available, using the same grouping technique, it was shown how an algorithm such as the one in [13] could be designed (where processors inside a group exchange messages through a cyclic path) with a message overhead of at most \(O(N \log^4 N)\). Both these algorithms are shown to have a much smaller message overhead than the algorithm in [13]. Numerical evaluation of the message overhead for specific values of different parameters show that in practice, the algorithm will work efficiently with a small number of messages.

### Acknowledgments

We wish to thank Alan Olson for providing us with the manuscript of [13] before it appeared in the proceedings.

### References


