Engineering Journal of Qatar University, Vol. 5, 1992, p. 145 - 162

RANDOM DISTRIBUTED ALGORITHMS FOR CLOCK SYNCHRONIZATION

Shoichiro Nakai*, Nasser Marafih** Shingo Fukui* and Satoshi Hasegawa*
*C&C Systems Research Laboratories, NEC Corporation, Japan.
**Qatar University, Doha, Qatar.

(First Received April 1992; accepted in revised form October 1992)

ABSTRACT

This paper proposes clock synchronization algorithms based on the idea in which randomly chosen m out of the total N processors cooperate to perform the clock adjustment of all processors in a distributed system. Selection of m processors is performed in a fully distributed way. Two types of algorithm which include different random mechanisms of processor selection are described. In the first algorithm, all processors adjust their clocks to the average of received clock signals issued by randomly chosen m processors. In the second algorithm, each processor chooses m processors randomly at its own will and adjust its own clock to the average of the chosen processor clocks. Fault tolerance against processor or link failure is taken into account in both algorithms. These algorithms exhibit desirable features in practical sense like simplicity in implementation, a small number of message exchanged, etc., so that the algorithms can be applied to sufficiently large system. Transient and steady state performances of the proposed algorithms are verified through simulation.

INTRODUCTION

Recently, clock synchronization in a loosely coupled distributed system has received substantial attention as one of the basic operating system functions or network control primitives. Processors cannot always keep in synchronization without any control. In order to enable processors to maintain clocks that are synchronized with one another at any time, periodical clock resynchronization mechanisms are needed. The problem is more complicated if there exist some faulty processors or communication failures. Clock sysnchronization plays an important role in many diverse applications like computer communication, real-time computing systems, data gathering and control and command systems.

Clock synchronization algorithms so far reported can be classified into two major categories, namely, a master-slave algorithm and a fully distributed algorithm. In the master-slave algorithms, all processors set their clocks to the value issued by a master processor. The master processor is not assigned ahead of time but rather elected (1) or agreed upon (2) according to some procedure. These procedures are distributed in the sense that any processor can be a master provided that all processors agree on that choice. In a fully distributed algorithms (3), all processors in the system exchange their clocks first, and then, each processor adjust its clock by using all the exchanged clocks.

In the master-slave algorithms, any processor may be required to become a master when a failure happens to the current master processor. In the TEMPO's algorithm implementation (1), an election mechanism is carried out using a time-out mechanism. The first processor whose timer expires after a failure has occurred in the current master processor becomes a candidate for the new master. After the candidate broadcasts an election message to all processors notifying them of its candidacy, a new master is elected among them. An autonomous recovery mechanism is embedded in a another algorithm (2). According to the algorithm, a processor which has the fastest clock becomes a master in each clock synchronization period. In these ways, the master-slave algorithms need some sort of mechanism to recover from a specific processor failure. In addition, the time taken to decide a winner master among contending candidates might be long, especially in a large system.

The fully distributed algorithm (3) focuses on fault tolerance against even malicious failures. In order to tolerrate the number of f malicious processors, $f+1$ rounds of message exchange are needed between all the processors to decide a new clock value. A message contains the clock value of the sender process at the instant of transmission. In the first round, all processors exchange their own private clocks by sending their clock values to all processes in the system. In the f succeeding rounds, all processors exchange other processor's clocks. These algorithms are not practical in a large system since a large number of message exchange is needed.

The clock synchronization algorithms in this paper are designed to have two main characteristics. First, distributed control is employed so that there is no need to consider a specific processor's failure. Second, simplicity is required so that the techniques apply even for large systems. That is, a practical algorithm should work with small number of message exchanged between processors for efficiency in utilizing communication resources. To meet the characteristics mentioned above, algorithm based on a random mechanism are proposed, in which a random selection of m processors out of N cooperate to adjust all clocks in the system. These m processors are to chosen in a fully distributed manner. Since a subset of all processors controls the adjustment of all clocks, these algorithms are referred to as *partially distributed algorithms* in this paper. Two types of partially distributed algorithms which include different random mechanisms of m processor selection are described and their performances evaluated in the following sections.

PARTIALLY DISTRIBUTED ALGORITHMS

Assumptions

Consider a geographically distributed system consisting of N processors and communication links connecting them. No restriction on the network topology is assumed. Let the set of processors and the corresponding clock times be $(P_1, P_2, \ldots\ldots, P_n)$ and $(C_1, C_2, \ldots\ldots, C_n)$, respectively. The following two conditions are assumed in this paper.

(A1) All clocks of non-faulty processors are assumed to be fairly accurate and run approximately at the same rate.

$$\left|\frac{d}{dt}C_i(t)-1\right|<\rho \tag{1}$$

where a fixed value ρ represents the maximum error in a clock's running rate against the real time t. The clocks are not necessarily synchronized at the initial execution of the algorithms.

(A2) Some restriction on the transmission delay is needed. More specifically the following condition is assumed if there are no failures in the system.

$$d_{i,r} < d_{max} \tag{2}$$

where $d_{i,r}$ is the transmission delay between P_i and P_r. The interval between two contiguous synchronization controls is assumed to be long compared with the maximum transmission delay, $d_{\max}$.

A processor or a communication link might be faulty during synchronization control. In fact, a faulty processor might report different clock times to different processors and also messages might not reach their destinations due to link failures. In this paper, the assumptions (A1) and (A2) are not always kept in the case of processor or communication link failures. In other words, if (A1) and/or (A2) are not satisfied, some kinds of failures are considered to have happened.

Algorithm Description

Algorithms for clock synchronization are desired to maintain all processor clocks to be synchronized within a fixed range. For any two non-faulty processors P_i and P_j, the following condition is satisfied.

$$\delta_{i,j}=\left|C_i(t)-C_j(t)\right|<\varepsilon \quad \text{for all } t, \tag{3}$$

where $\varepsilon << 1$.

Generally speaking, this condition could be fairly hard to satisfy especially in the case of processor or communication link failures. The proposed algorithms ensure maintenance of the condition mentioned above in a probabilistic manner even in faulty condition, due to a random mechanism in the procedures. Actually, the algorithms could be controlled to keep all clocks within a fixed range at almost all times in a real system.

Algorithm 1: Passive Random Distributed Algorithm (PRDA)

As discussed in the previous section, the proposed algorithms fall in the partially distributed category. In the first algorithm, m out of the total N processors, which are chosen randomly, broadcast their clocks to all processors including themselves. Since m processors are chosen randomly and all processors adjust their clocks according to the messages from m processors, this algorithm is referred to as *Passive Random distributed Algorithm (PRDA)*. Fig. 1 shows the algorithm

```
loop (k=0;  ;++k)
        select
        /**** Message Transmission Phase ****/
                clock interrupt ($C_i = kT_s$)
                R← Random Number Generator
                if (R < m/N)
                        send ($C_i$ to all Processors)

        /**** Message Reception Phase ****/
                receive ($C_j$ from $P_j$)
                if ($C_j$ is only one message from $P_j$)
                        $d_{j,i} \leftarrow C_j - C_i$
                        save buffer ($d_{j,i}$ in $P_i$'s buffer)

        /**** Clock Adjustment Phase ****/
                clock interrupt ($C_i = kT_s + T_c$)
                read buffer ($d_{jl,i}$ ($l=1,..,m'$) in $P_i$'s buffer)

        /** Check Legitimacy of Saved Readings **/
                for ($l=1$; $l \leq m'$; ++l)
                        for ($g=1$; $g \leq m'$; ++g)
                                if ($g \neq l$)
                                        $d_{jl,jg} \leftarrow d_{jl,i} - d_{jg,i}$
                        if (all $d_{jl,i} > e$)
                                $d_{jl,i} \leftarrow 0$

        /** Adjustment for Clocks **/

                $d_i \leftarrow \frac{1}{m'} \sum_l \delta_{jl,i}$
                $C_i \leftarrow kT_s + T_c + d_i$
        end select
end loop
```

Fig. 1: The Passive Random Distributed Algorithm (PRDA) Performed by Processor P_i.

executed by P_i during the k-th synchronization period $[kT_s, (k+1)T_s]$, where T_s is the synchronization period. The algorithm can be divided into three phases, namely, a message transmission phase, a message reception phase and a clock adjustment phase. First, in the message transmission phase, P_i, at kT_s on its own clock, P_i decides whether it must broadcast its clock value to all processors or not. This procedure is realized based on random selection mechanism. That is, P_i generates a pseudo-random number R, which is distributed uniformly from 0.0 to 1.0 and broadcast it clock signal to all the processors if R is less than m/N. The value m stands for the expected number of processors which can be selected to broadcast their clocks, and N is the total number of processors. The desired number of processors needed to synchronize all clocks is controlled by choosing m. Second, in the message reception and the adjustment phases, P_i, normally receives all the m messages and stores the differences between the received clock times and C_i. When $C_i = k\ T_s + T_c$, where T_c is set to be sufficiently large value compared to d_{max}, P_i adjusts C_i and the average of the stored differences. Since the number of received messages may be less than m when processor or link failures occur, P_i uses m' $(m' \leqslant m)$ messages which are received until the time $C_i = k\ T_s + T_c$.

In order to tolerate processor failure, a legitimate received clock value must conform to the following conditions:

(C1) The selected processor issues only one message in a synchronization period. If more than one message is received from the same sender during a synchronization period, these messages are discarded and the sender is assumed to be faulty.

(C2) The clock difference between two non-faulty processor clocks must be within ε. Hence, the received clock count differing from all other clock counts by more than ε should be discarded.

Next, it will be shown by an example how the algorithm will succeed in bringing all clocks closer to each other. In this example, both the transmission delay and clock drift which will be considered in the next section are neglected. Let $N = 5, T_s = 1$ hour, $T_c = 0.5$ hour and $m = 2$. Examples of clock readings along with the random number R are as shown in Fig. 2a. According to the selected R in this example, only processors P_2 and P_4 will succeed in broadcasting their clocks since their R's are 0.2 and 0.3, respectively. P_4 first transmits its clock at 1:00 to all processors including itself. Upon receiving this message, every processor saves the differences of the received clocks from its own. The same procedure is performed when P_2 transmits it clock at 1:00. When each processor reads 1:30 on its clock, it calculates the average of the two readings stored in the buffer and adjusts the clock by adding its clock and the calculated average value as shown in Fig. 2b. For example, P_5 first adjusts its clock to 1:26. Fig. 2c shows all the adjustment values computed by all the processors. In this figure, P_2 through P_5, should have already clock, it chooses m distinct processors randomly. After choosing m processors, P_i sends a message to each one of them requesting their clocks. Second, in the message reception and the clock adjustment phase, P_i takes the difference between responses and its own clock and store this value in its buffer when it receives a reply. At the time when P_i reads $kT_s + T_c$, it computes the adjustment value by taking the average of m' $(m' \leq m)$ saved differences and adds this value to its own clock. The number of received messages, m', may be less than m if there exist some faulty processor or communication failures. The same legitimate check as in the case of PRDA is performed in order to exclude possible processor or link failures. Finally in the reply phase, when P_i receives a request from another processor, P_i must reply by sending a message containing P_i's current clock. In ARDA, each processor chooses m distinct processors in an autonomous and random way, so that each clock may be adjusted by using different clocks initiated by different subset of processors. Hence, it would take more than one synchronization period for ARDA to reach convergence.

An example showing how this algorithm works is given in Fig. 4. In this example both the transmission delay and clock drift are neglected as in PRDA. Let $N = 5$, $T_s = 1$ hour, $T_c = 0.5$ hours and $m = 2$. An example of clock readings along with chosen processors is shown in Fig. 4a. These processors are chosen when each one of the processor clocks strikes 1:00, 2:00, ...etc. Taking the processor P_5 at 1:00 for instance, P_5 selects P_2 and P_4, and sends requests to them in order to read their clocks. The requested processors P_2 and P_4 reply by transmitting their clocks to P_5 which at that instant have clock readings of 0:54 and 0:58, respectively. Upon receiving these messages at 1:00, P_5 computes the differences $\delta_{2,5}$ = 0:54-1:00 = -0:06 and $\delta_{4,5}$ = 0:58-1:00 = -0:02. At the instant when P_5 clock strikes 1:30, P_5 calculates the average of the differences between the received clocks and its own clock which gives δ_5 = (-0:06-0:02)/2 = -0:04 and adjusts its clock value by that value. Fig. 4b shows the adjustment values computed by all processes. Fig. 4c shows the clock readings after the adjustment is performed.

Analysis

In this section, the analytical characteristics of the proposed algorithms are discussed. The clock difference between two arbitrary processors and its variance is evaluated in this analysis. The convergence condition and convergence time of ARDA will also be derived. Before the analysis of each algorithm, general equations and assumptions are made in the following.

The processor P_i's clock C_i is assumed to be expressed as a linear function of the reference time t_k, which is shown in Fig. 5.

$$C_i(t_k) - kT_s = (1-\beta_{i,k})\{t_k - (kT_s + \Delta_{i,k})\} \quad \text{for } kT_S \leq t_k < (k+1)T_S \tag{4}$$

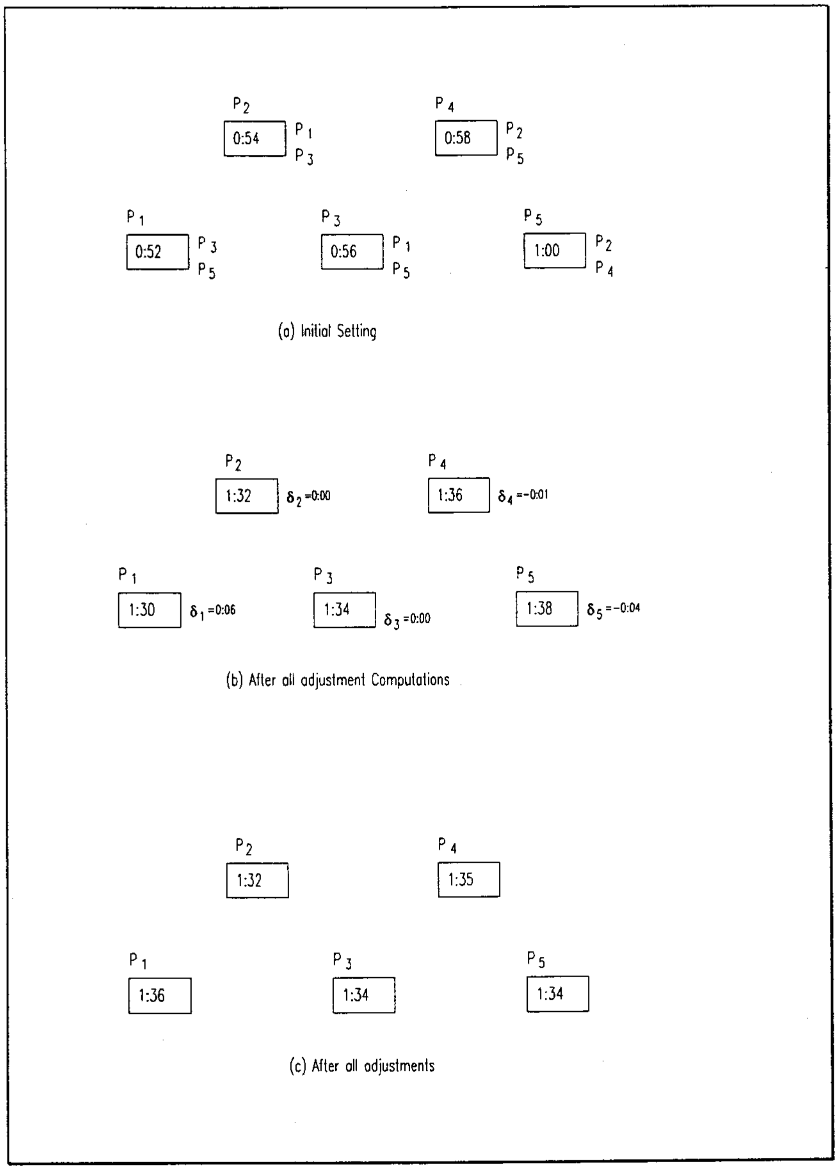


Fig. 4: Adjustment Example in ARDA.

The reference time t_k is defined during the *k-th* synchronization period. The value $\beta_{i,k}$ is the drift of C_i against the reference time t_k, that is, the error in a C_i's running rate, and $\Delta_{i,k}$ stands for the offset value of C_i from t_k at $C_i\ (t_k) = kT_s$. The specific definition of the reference time t_k is described later for both algorithms respectively. When P_i reads $C_i\ (t_k) = kT_s$, another processor P_j reads C_j on its own clock as follows.

$$C_j(t_k)\Big|_{t:C_i=kT_s} = kT_s + (1-\beta_{j,k})(\Delta_{i,k} - \Delta_{j,k}) \tag{5}$$

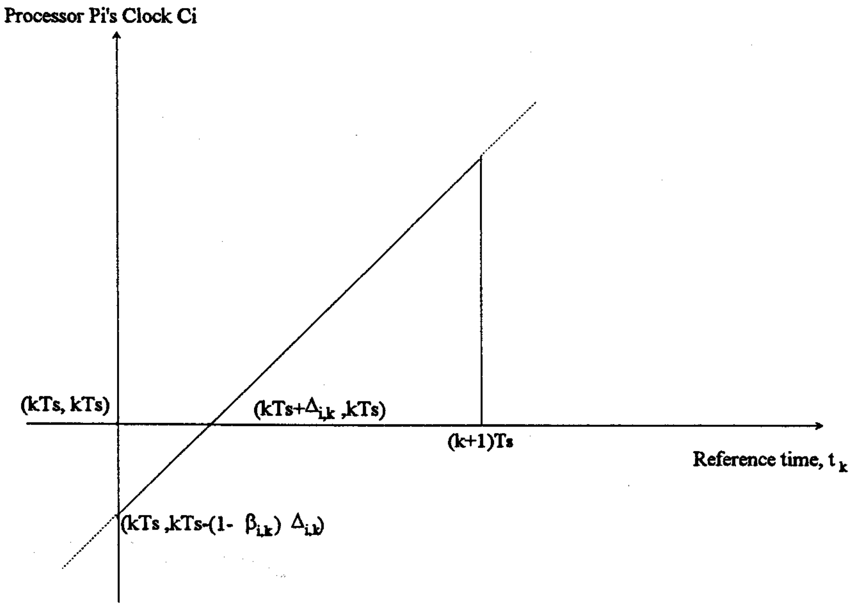


Fig. 5: Schematic Expression of C_i.

From equation 5 and the transmission delay from P_i to P_j, the clock difference between C_i and C_j at the instance when P_j is aware that P_i has read kT_s, is represented as follows:

$$\delta_{i,j} = (1-\beta_{j,k})(\Delta_{i,k} - \Delta_{j,k}) + d_{i,j} \tag{6}$$

Where $d_{i,j}$ stands for the time taken for a message issued by P_i to arrive at P_j.

Basically, the handling of the reference time t_k and $\delta_{i,j}$ is different between PRDA and ARDA. The analysis for both algorithms is explained in the following subsections.

i) PRDA

In PRDA, P_j decides whether it must send C_j to all processors when P_j reads kT_s. As a result, m processors are selected and succeed in sending their clocks, where the number m denotes the expected value of the number of successful processors. A receiving processor P_i takes the average of m clocks issued by the chosen m processors and adjusts its C_i. Since the instant when the clock readings are made is determined by a sending processor P_j in PRDA, $-d_{j,i}$ in equation 6′ is applied to obtain the clock adjustment value.

$$-\delta_{j,i} = (1-\beta_{i,k})(\Delta_{i,k} - \Delta_{j,k}) - d_{j,i} \tag{6'}$$

The processor P_i adjusts its clock by adding the average of m clock differences at the time when C_i strikes $kT_s + T_c$. The P_i's clock $C_i'(t_k)$ after adjustment during k-th synchronization period can be expressed as follows:

$$C_i'(t_k) - \left[kT_s + \frac{1}{m}(1-\beta_{i,k}) \sum_j \left\{ (\Delta_{i,k} - \Delta_{j,k}) - d_{j,i} \right\} \right]$$
$$= (1-\beta_{i,k})\left\{ t_k - (kT_s + \Delta_{i,k} \right\} \quad \text{for } kT_S + T_C \le t_k < (k+1)T_S \tag{7}$$

Since the reference time t_k in PRDA is defined as the average clock of the selected m processors, the value $\Sigma_j \Delta_{j,k}$ is equal to zero.

As result, the difference between t_k and C_i measured at $t_k = (k+1)\,T_s$ is obtained easily as follows.

$$\begin{aligned} \Delta_{i,k+1} &= \beta_{i,k} T_s + (1-\beta_{i,k}) d_i \\ &= \beta_{i,k} T_s + d_i^{(k)} \end{aligned} \tag{8}$$

Where d_i is defined as $\Sigma_j\, d_{j,i}/m$. In this equation, $(1-\beta_{i,k})\, d_i$ which is represented as $d_i^{(k)}$ is a delay time measured by the processor P_i's clock. The expected value of $d_i^{(k)}$ for all processors is assumed to be zero in terms of the reference time t_k. Also the statistical value of the clock drift, $\beta_{i,k}$, are independent of the reference time t_k. It can be easily seen that the variance of the clock differences between arbitrary two processors $\sigma^2\ \delta_{PRDA}$ is written as follows:

$$\sigma_\delta^2 \Big|_{PRDA} = 2(\sigma_\beta^2 T_s^2 + \sigma_d^2) \tag{9}$$

Where σ_β^2 *and* σ_d^2 are the variances of the drift and the transmission delay, respectively. This equation shows that the variance of the clock differences, $\sigma_\delta^2 \,|_{PRDA}$, does not depend on the initial condition and the synchronization period k. In a real situation, the transmission delay would dominate the synchronization precision attained. The value of σ_d^2 could be small when m is chosen to be large.

ii) ARDA

In ARDA, when the processor P_i reads kT_s, it requests randomly selected m processors to send their clock values. Since the instance when the clock readings are made is determined by a receiving processor in ARDA, $\delta_{i,j}$ represented in equation 6 is used for clock adjustment value. Since the reference time t_k of ARDA is defined as the average value of all the processor clocks in the k-th synchronization period, the drift of C_i is expressed by β_i instead of $\beta_{i,k}$. The P_i's clock $C_i'(t_k)$ after adjustment during k-th synchronization period can be given by

$$C_i'(t_k) - \left[kT_s + \frac{1}{m}\sum_j \left\{ (1-\beta_j)(\Delta_{i,k} - \Delta_{j,k}) + d_{i,j} \right\} \right]$$
$$= (1-\beta_i)\left\{ t_k - (kT_s + \Delta_{i,k} \right\} \text{ for } kT_S + T_C \le t_k < (k+1)T_S \tag{10}$$

The difference between t_k and C_i at $t_k = (k+1)T_s$ is obtained easily by substituting $(k+1)T_s$ into t_k in equation 10 as follows:

$$\Delta_{i,k+1} = \beta_i T_s - d_i^{(k)} - \beta_i \Delta_{i,k} + \frac{1}{m}\sum_j \left\{ (1-\beta_j)\Delta_{j,k} + \beta_j \Delta_{i,k} \right\} \tag{11}$$

According to the definition of reference time of ARDA, the expected value of $\Delta_{i,k}$ is equal to zero. The expected value of delay is assumed to be zero in terms of the reference time t_k, and β_i and $\Delta_{i,k}$ are assumed to be mutually independent variables. From equations 9 & 11, the variance of clock differences between arbitrary two processor clocks at $(k+1)T_s$ instance can be expressed recursively as:

$$\sigma^2_{\delta_{k+1}} = \sigma^2_\delta \big|_{PRDA} + \sigma^2_{\delta_k} \left\{ \sigma^2_\beta + \frac{1}{m}(1 + 2\sigma^2_\beta) \right\} \tag{12}$$

The convergence condition in term of m, and converged variance can be derived as Equation (13) and (14), respectively

$$m > \frac{1 + 2\sigma^2_\beta}{1 - \sigma^2_\beta} \tag{13}$$

$$\sigma^2_{\delta_\infty} \big|_{ARDA} = \sigma^2_\delta \big|_{PRDA} \cdot \frac{m}{(1-\sigma^2_\beta)m - (1 - 2\sigma^2_\beta)} \tag{14}$$

It is easily found that this algorithm can converge at $m = 2$ if σ^2_β is negligibly small. Normally, this condition is easily realized when a crystal clock is employed in each processor. In case that m is sufficiently large, equation 14 is approximately represented as

$$\sigma^2_{\delta_-}\Big|_{ARDA} = \sigma^2_{\delta}\Big|_{PRDA} \cdot \frac{1}{1-\sigma^2_{\beta}} \tag{15}$$

This equation indicates that the converged variance in ARDA is $1/(1-\sigma^2_\beta)$ times larger than that in PRDA, however, variances in both PRDA and ARDA are almost the same if σ^2_β is sufficiently small.

Suppose that the algorithm has converged if $\sigma^2_{\delta_{k+1}} - \sigma^2_{\delta_k} < \gamma$ where γ is sufficiently small number. From equation 12 the number of required synchronization periods to reach convergence in ARDA is obtained as follows

$$k > \frac{\log\gamma - \log(\sigma^2_{\delta_0} - \sigma^2_{\delta 1})}{\log\left\{\sigma^2_\beta + \dfrac{1+2\sigma^2_\beta}{m}\right\}} + 1 \tag{16}$$

In case that $m >> 1$ and $\sigma^2_\beta << 1$, the above equation is approximated as

$$k > \frac{\log\sigma^2_{\delta_0} - \log\gamma}{\log\{m\}} + 1 \tag{17}$$

equation 17 means that the convergence time depends on the initial variance of clock differences, $\sigma^2_{\delta_0}$, and the chosen number m.

Through the analysis and qualitative nature of both algorithms, differences between PRDA and ARDA are clarified. One of the basic differences between them is the way in which clock readings are made. In PRDA, randomly selected processors autonomously broadcast their clocks to all processors, while in ARDA, a processor obtains another processor clock reading by sending a request to that processor. Table 1 lists the main differences between the two algorithms.

Table 1
Differences between PRDA and ARDA

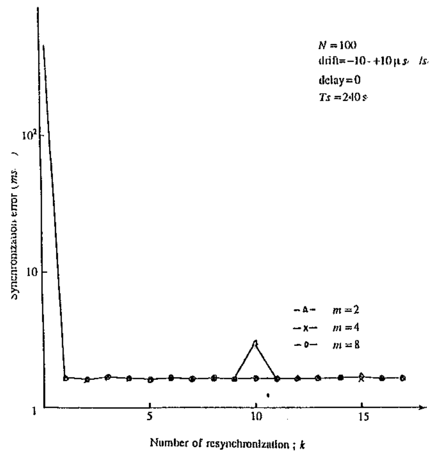
Property	PRDA	ARDA
Nature	passive	active
Message Complexity	$(N-1) \times m$	$2 \times N \times m$
Preferred Network	broadcast	point-to-point
Convergence Time	one period	$\frac{\log\sigma^2\delta_0 - \log\gamma}{\log m} + 1$ periods
Number of Selected Processors	random number	fixed number

SIMULATION

The proposed algorithms have been simulated in order to investigate the effect of random selection on the overall performance of the system. The performance measure used is the synchronization error. The synchronization error is defined as the average absolute value of clock differences between all pairs of processors in the system, whose statistical values are hard to drive analytically, has been obtained in both algorithms. A process is created for each process in this simulation, which maintains the important parameters related to that processor such as clock time, identifier and others that are needed by the operating system for housekeeping. The simulation program is written in C language under UNIX 4.3 BSD operating system. Due to restrictions and limitations imposed by the UNIX operating system in dealing with processes, a small kernel to handle the creation and switching of processors was developed. Basically, when the time has come to execute a predefined procedure in each process, the process is invoked by the kernel. In this sense, the simulation program is categorized as an event driven type. A total of N processes, where N is the number of processors in the system is executed concurrently according to the order from the kernel. In this simulation, three important parameters which affect the performance of the algorithms are considered, namely, the number of randomly chosen processors, m, clock shift, β_i, and transmission delay, $d_{i,j}$. Through the simulation, the total number of processors, N, is set to be 100.

As mentioned earlier, the proposed algorithms do not require the processors to be synchronized in strict sense at the initial state. The main concern here is to observe the transient behavior of both algorithms, that is, to see the number of periods needed to bring the processor clocks within the required precision. Fig. 6 & 7 show these behaviors, where the synchronization period, T_s, is equal to 240 seconds, and the initial offsets and the clock drift are distributed uniformly in the range of ∓ 1 s and ∓ 10 μs/s, respectively. Fig. 6 shows the result in case that no communication delay exists and Fig. 7 shows the effect of delay where delay is distributed uniformly from 10 to 30 ms. In case of PRDA, it takes exactly one period to reach the presision needed, as was anticipated due to the passive nature of the algorithm see Fig. 6a & 7a. In Fig. 6a, a little bit of unstable behavior can be observed in case that $m = 2$ and $k = 10$. A possible reason is that the number of processors initiating a message is given by an estimated value instead of fixed number in PRDA. This causes the probability that no processor issues a clock value in some synchronization period to be relatively large if m is small, which leads processor clocks to drift apart. In Fig. 7a which considers the transmission delay, resultant synchronization errors vary roughly when m is small. This is mainly due to the fact that the averaging gain for the delay factor cannot be obtained if m is small.

On the other hand, due to the active nature of ARDA, it is easy to see from Fig. 6b & 7b that it would take more than one period for the algorithm to reach convergence. It can be also observed that synchronization error would be reduced by choosing a larger size of *m*. The convergence time needed for ARDA is obtained both from the simulation result in Fig. 6b and the analysis in equation 17, which is shown in Table 2. In order to get the analytical result, γ in equation 17 is assumed to be 10^{-6} and all other parameters are set to be the same as those in the simulation case. Table 2 indicates that the simulation and analytical results show a fairly good coincidence.



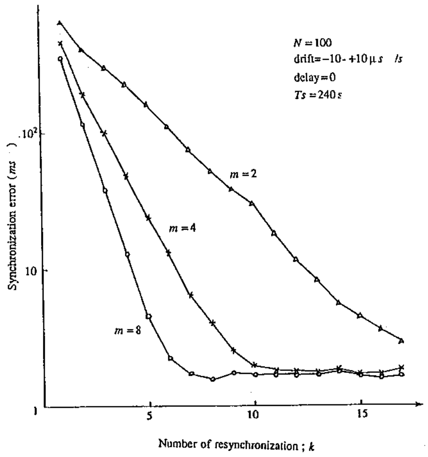
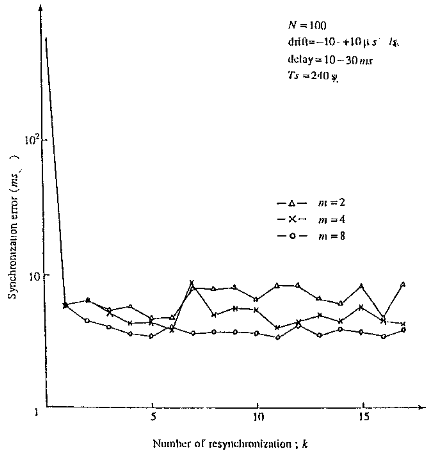


Fig. 6a: Transient Behavior in PRDA.
Fig. 6b: Transient Behavior in ARDA.



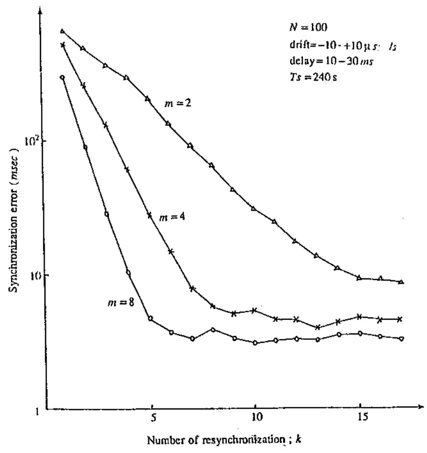


Fig. 7a: Transient Behavior in PRDA.
Fig. 7b: Transient Behavior in ARDA.

Table 2
Convergence Periods (k) for Simulation and Analytical Results

m	*Simulation*	*Analysis*
2	22	19.3
4	11	10.2
8	7	7.1

Fig. 8a & 8b show the synchronization error versus various synchronization intervals, T_s, in the steady state after convergence. The distribution of clock drift and the transmission delay are the same as in the previous case. These figures show that the resultant synchronization error depends on the value *m* in both algorithms. It is important to notice that both algorithms perform better when the size of *m* is allowed to become larger. The reason is that the effect of the transmission delay is surely reduced by the averaging operation as the value of *m* increases. A good choice for *m* is considered to be larger than 4. According to the results in Fig. 8, PRDA and ARDA attain approximately the same precision especially for a large number of *m*. Another interesting feature of the algorithms can be seen from Fig. 9, where increasing the number of processor in system would have negligible effect on the synchronization error. Hence, adding new processors in the system will not require any modifications.

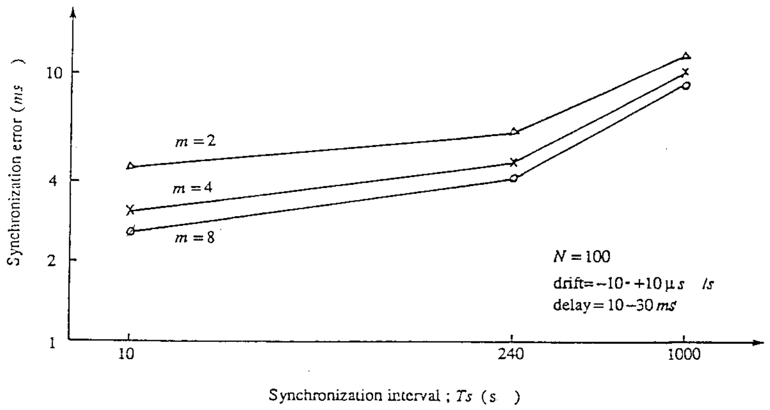


Fig. 8a: Synchronization error vs. Synchronization interval in PRDA.

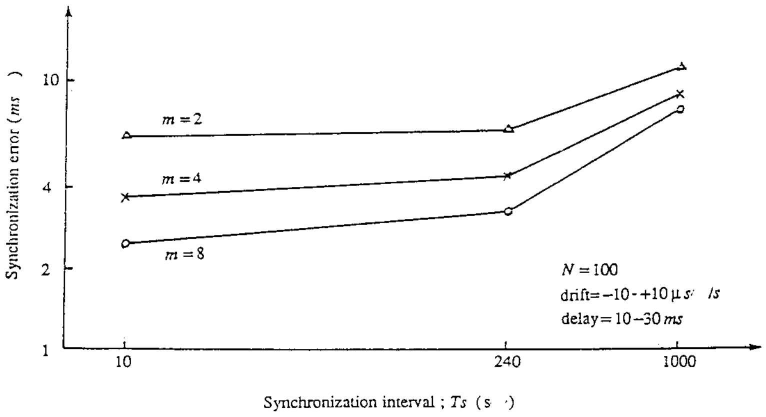


Fig. 8b: Synchronization error vs. Synchronization interval in ARDA.

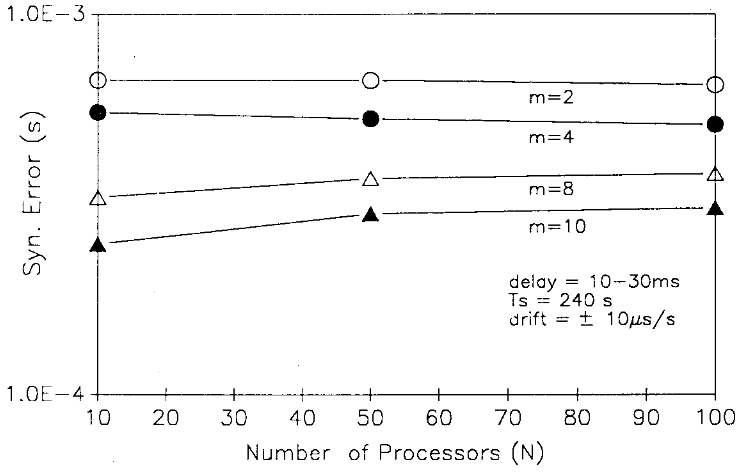


Fig. 9: Synchronization error vs. Number of Processors (N).

CONCLUSION

Two simple practical algorithms with a random mechanism for synchronizing clocks have been proposed. Unlike conventional algorithms, these algorithms are based on a random selection of the master processors in an active or passive way. As a result, the proposed algorithms can reduce the number of messages exchanged and also avoid relatively complicated procedures to provide a high degree of fault tolerance. They also exhibit many desirable features like distributed control, ease of implementation, fast in execution, network traffic efficiency, etc. Although both algorithms are similar in concept, they are different in implementation. The first one, called PRDA, has a passive nature where all processors receive the same *m* clocks values from randomly selected *m* processors in a distributed fashion. In ARDA, each processor request *m* randomly chosen processors to sent their clocks in an autonomus way, which shows the active nature of the algorithm. The most important design parameter of these algorithms is the selection of *m*. The size of *m* will have an affect of precision, stability and failure tolerance of the algorithm. Simulation results clarify the characteristics of the algorithms for various values of *m*. The result shows that *m* should be selected to be larger than 4. The applied area of the algorithms depends primarily on the nature of the network employed. PRDA is suitable in the broadcast network environment while ARDA is fitted to the point-to-point network.

ACKNOWLEDGEMENT

The authors would like to thank K. Watanabe and A. Kanemasa for their helpful suggestions and encouragement.

REFERENCES

1. **Gusella, R.** and **Zatti, S., 1986,** An Election Algorithm for a Distributed Clock Synchronozation Program, Proceedings of the 6th International Conference on Distributed Computing System, p. 364-371.

2. **Cristian, F., Aghili, H.** and **Strong, R., 1986,** Clock Synchronization in the Presence of Omission and Performance Faults and Processor Joins, Proceedings of the 16the Annual International Symposium on Fault-Tolerant Computing, p. 218-223.

3. **Lamport, L.** and **Melliar-Smith, M., 1985,** Synchronizing Clocks in the Presence of Faults, Journal of the Association of Computing Machinery, vol. 32, no. 1, January, p. 52-78.