1994 IEEE Workshop on Fault-Tol. Par. and Dist. Syst., College Station, TX, June 13-14. (Appears in Fault-Tol. Par. and Dist. Syst., *D. Pradhan and D. Avresky, eds., IEEE Comp. Soc. Press, pp. 268-277.)*

Fault-Tolerant Clock Synchronization for Distributed Systems with High Message Delay Variation*

Marcelo Moraes de Azevedo and Douglas M. Blough
Deparment of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92717

Abstract

Fault-tolerant clock synchronization is an important requirement in many distributed systems, especially in time-critical and safety-critical applications. Frequently, interactive convergence algorithms are used for fault-tolerant clock synchronization, providing advantages such as fully distributed operation, low message exchange overhead, and simplicity of implementation. This paper presents the measured performance of three interactive convergence clock synchronization algorithms. Our experiments were conducted in a distributed UNIX environment featuring high message delay variation, which poses severe constraints on the clock synchronization tightness that may be achieved. The algorithms that were tested are: FTMA (fault-tolerant midpoint algorithm) [1], AEFTMA (adaptive exponential averaging fault-tolerant midpoint algorithm) [2], and SWA (sliding window algorithm) [3]. Our experimental results indicate that SWA outperforms the other algorithms in this environment, being able to achieve tighter synchronization under different simulated fault conditions. The superiority of SWA can be attributed to its high degree of fault tolerance, combined with its ability to treat messages with much longer than expected delays as faults.

1: Introduction

In distributed systems, computers cooperate to provide the expected functionality to a given application. Some tasks that are often found in such systems are: synchronizing activities that occur at different points of the system, ordering events in time, enforcing deadlines, and measuring elapsed time. A system with one or more of these requirements must use proper synchronization mechanisms to establish an agreed-upon global time scale among its components. Particularly in the case of safety-critical applications, synchrony must be maintained in spite of the presence of faults in the system.

Frequently, fault-tolerant clock synchronization is achieved via *interactive convergence algorithms* in which nodes exchange their clock values and determine clock correction terms at regular intervals. This paper presents the measured performance of three interactive convergence algorithms: the sliding window algorithm (SWA) [3], the fault-tolerant midpoint algorithm (FTMA) [1], and the adaptive exponential averaging fault-tolerant midpoint algorithm (AEFTMA) [2]. The measurements were carried out with an application-level implementation running in a distributed UNIX environment [2]. This environment poses some practical constraints to clock synchronization, in particular a high variation in the

*This research was supported in part by Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq - Brazil), under Grant 200392/92-1, by the National Science Foundation under Grant CCR-9318495, and by the California Space Institute under Grant CS-54-92.

message delay. Our results are therefore representative of a broad class of systems where a low variation in the message delay cannot be achieved. Our experimental results indicate that SWA maintains tighter clock synchronization than the other algorithms. We will show that this results from SWA's higher degree of fault tolerance [3], together with its ability to treat messages with much longer than expected delays as faults.

2: Background: Interactive convergence algorithms

In a distributed system, time is usually observable in a local reference referred to as *clock time*. Clock times may differ both in absolute value and in rate from an assumed Newtonian time frame, which is not directly observable and is referred to as *real time*.

The clock of a node can be represented by a mapping C from real time to clock time. For example, $C_i(t) = T$ means that at real time t the clock of node i has value T. Clocks in different nodes in a distributed system tend to drift apart from each other with the passage of time, because they typically do not tick at exactly the same rate. This brings about the need of some form of clock synchronization scheme in systems where a global time scale must exist. Moreover, clock synchronization must often be achieved in environments in which challenges such as faults and high message delay variation may exist.

Interactive convergence is a widely used approach to fault-tolerant clock synchronization, being suitable for a large range of distributed systems applications. Clock synchronization is accomplished in a fully distributed fashion, allowing every node to exhibit equal functionality with regard to synchronization. No single point of failure exists, and expensive modules to provide real-time clock references are not required. Interactive convergence algorithms are driven by the multiple clock sources normally found in distributed systems, and therefore can be implemented at little additional cost. In addition, such algorithms usually feature low message exchange overhead, requiring only $O(n)$ messages per round in an n-node system with broadcast capabilities. Interactive convergence algorithms provide *internal clock synchronization*, meaning that the system's global time scale is not necessarily synchronized to an external time reference.

In an interactive convergence algorithm, nodes periodically exchange synchronization messages containing their clock values and then use the received values to adjust their clocks. The interval of time between successive resynchronizations is dubbed a *resynchronization interval* or a *round* of the algorithm and its duration is denoted by T_{int}. A node that is exchanging messages for the rth time since the clock synchronization process was started is said to be in its rth resynchronization interval. One technique that can be used to exchange clock values is broadcasting, which is in fact the method used in our experiments.

The process by which a node assigns a clock value to be sent in an outgoing message or to be attached to an incoming message is referred to as *time-stamping*. The clock value that is sent in a synchronization message is referred to as the *sender's time-stamp* (T_{send}). Similarly, the clock value that is attached to an incoming synchronization message is referred to as the *receiver's time-stamp* (T_{rec}).

Assume that at real time $t^r_{send}(i)$ node i time-stamps its rth synchronization message with a sender's time-stamp $T^r_{send}(i)$. Let the total message delay between nodes i and j during the rth resynchronization interval be $d_r(i,j)$, such that i's message is time-stamped by j at real time $t^r_{rec}(i,j) = t^r_{send}(i) + d_r(i,j)$. If j's clock reads $T^r_{rec}(i,j)$ at real time $t^r_{rec}(i,j)$, then the clock value of i can be estimated by j at real time t^r_j, $t^r_j \geq t^r_{rec}(i,j)$, by $C^r_{i,j}(t^r_j) = T^r_{send}(i) + d^r_{med}(i,j) + (C_j(t^r_j) - T^r_{rec}(i,j))$, where $d^r_{med}(i,j)$ is j's estimate for the message delay between nodes i and j during the rth resynchronization interval.

The *clock deviation* between nodes i and j at real time t is $C_j(t) - C_i(t)$. The clocks

of any two nonfaulty nodes a and b in the system deviate by at most Δ_{int} seconds at any real time $t \geq 0$, where Δ_{int} is referred to as the *worst-case clock synchronization tightness*. Formally, $\forall a, b,\ \forall t \geq 0,\ |C_a(t) - C_b(t)| \leq \Delta_{int}$.

The *estimated clock deviation* between nodes i and j as measured by node j at real time t_j^r during the rth resynchronization interval is given by

$$\Delta_{i,j}^r = C_j(t_j^r) - C_{i,j}^r(t_j^r) = T_{rec}^r(i,j) - T_{send}^r(i) - d_{med}^r(i,j) \quad (1)$$

To simplify the terminology of this paper, we hereafter use the term *clock deviation* to refer to a *clock deviation estimate*. Whenever appropriate, we will remind the reader that a distinction exists as defined above.

At the end of the rth resynchronization interval, each node j, $1 \leq j \leq n$, will have collected a vector of clock deviations $\overline{\Delta_j^r} = [\Delta_{1,j}^r\ \Delta_{2,j}^r\ \cdots\ \Delta_{j-1,j}^r\ 0\ \Delta_{j+1,j}^r\ \cdots\ \Delta_{n,j}^r]$ (note that $\Delta_{j,j}^r = 0$). A clock correction term $CORR_j^r$ is then calculated from $\overline{\Delta_j^r}$ by means of a convergence function. Following this step, the local clock of node j is adjusted by $CORR_j^r$.

To guarantee that an interactive convergence algorithm maintains synchronization requires that clock synchronization messages be exchanged with some bounded delay d_{max}. Message delay is expected to vary in the range $[d_{min}, d_{max}]$, with $d_{var} = d_{max} - d_{min}$ being referred to as the *message delay variation* and $d_{var}/2$ being referred to as the *reading error*. Naturally, d_{var} affects the worst-case clock synchronization tightness (Δ_{int}).

Lundelius and Lynch [4] proved that in a system with n clocks and message delay variation d_{var} the clock synchronization tightness cannot be better than $d_{var}(1 - 1/n)$. Such a lower bound holds under the strong assumptions that all clocks run at a perfect rate and that there are no failures in the system. Under the same assumptions and also assuming that clocks initially have arbitrary values, Lundelius and Lynch give a simple algorithm that achieves this bound. Clearly, this lower bound also holds for the more realistic case that we consider in which clocks drift and faults occur.

Our implementation performs time-stamping of messages in the application level, using UNIX system calls for that purpose. This mechanism exhibits a highly variable delay between the moment a time-stamp is obtained and the corresponding message is actually sent or received. The resulting message delay variation in an application-level software implementation is typically several hundreds of milliseconds [2]. According to the Lundelius and Lynch lower bound, the worst-case clock synchronization tightness (Δ_{int}) should therefore be on the same order of magnitude.

An interesting aspect of implementing a clock synchronization algorithm in the application level is that a high independence of the hardware platform is obtained. Therefore, this approach can be readily used in existing systems.

Application-level clock synchronization can be used in many applications that do not require tight synchronization. Nevertheless, we would still like to have the best tightness that can be achieved in this environment. Considering this goal, our experiments evaluated three different interactive convergence functions: FTMA, AEFTMA, and SWA.

3: Description of convergence functions

As described in the previous section, interactive convergence algorithms exhibit great procedural similarity, with variations being limited to the convergence function that is used to compute the clock correction term at every round. In the remainder of this section, we describe the convergence functions used by FTMA, AEFTMA, and SWA.

3.1: Fault-tolerant midpoint algorithm

The fault-tolerant midpoint algorithm (FTMA) [1] relies on the hypothesis that at most k clocks are faulty at any resynchronization interval. At least $n = 3k+1$ nodes are required in the system to tolerate k Byzantine faults.

Assume that, at a given round r of the algorithm, a sorted clock deviation vector $\overline{X_j^r} = [x_1\ x_2\ \cdots\ x_i\ \cdots\ x_n]$, $x_i \leq x_{i+1}$, is available at node j (note that the elements in $\overline{X_j^r}$ are exactly the elements in $\overline{\Delta_j^r}$, except that in $\overline{X_j^r}$ they are sorted). To find the clock correction term, FTMA discards the k lowest and the k highest clock deviations and computes the arithmetic mean of x_{k+1} and x_{n-k}; that is, $CORR_j^r = (x_{k+1} + x_{n-k})/2$.

3.2: Adaptive exponential averaging fault-tolerant midpoint algorithm

The adaptive exponential averaging fault-tolerant midpoint algorithm (AEFTMA) [2] is a variation of the FTMA algorithm [1]. Assume that $CORR_{FTMA}^r$ is the correction term computed via the standard FTMA convergence function (i.e., $(x_{k+1} + x_{n-k})/2$) in the rth resynchronization interval and that $CORR_{AEFTMA}^{r-1}$ is the correction term computed by AEFTMA in the previous round. Instead of using $CORR_{FTMA}^r$ as the correction term during the rth resynchronization interval, AEFTMA computes its correction term by $CORR_{AEFTMA}^r = \beta(r) \cdot CORR_{FTMA}^r + (1 - \beta(r))CORR_{AEFTMA}^{r-1}$.

$\beta(r)$ is a weight factor in effect during the rth resynchronization interval, such that $0 \leq \beta(r) \leq 1$. The weight factor "smooths" the clock correction term $CORR_{AEFTMA}^r$, introducing a lagging mechanism between adjacent rounds. This approach follows the assumption that correction terms of similar magnitude are expected at every resynchronization interval, since changes in the drift rate of clocks are generally caused by intrinsically slow phenomena such as temperature variation and component aging. Therefore, excessive correction terms resulting from large message delay variation are attenuated by exponential averaging. Nevertheless, in order to guarantee a fast recovery of synchrony in case of transient fault occurrence, the weight factor varies adaptively according to the absolute value of the clock correction term computed by AEFTMA in the current round ($|CORR_{AEFTMA}^r|$).

The adaptive control of the weight factor must be chosen according to the expected message delay variation, which in an application-level implementation is highly dependent on aspects such as CPU and network utilization and the priority assigned to the clock synchronization process. We used in our implementation a 4-level stepwise function $f(|CORR_{AEFTMA}^r|)$, which selects the next-round weight factor $\beta(r+1)$ from a set of values $B = \{0.1, 0.25, 0.5, 1\}$. Selection of $\beta(r+1)$ is accomplished by comparing the current-round absolute clock correction term $|CORR_{AEFTMA}^r|$ with a set of thresholds $C = \{50$ ms, 100 ms, 150 ms$\}$, such that $\beta(r+1) = 0.1$ if $|CORR_{AEFTMA}^r| \leq 50$ ms, $\beta(r+1) = 0.25$ if 50 ms $< |CORR_{AEFTMA}^r| \leq 100$ ms, $\beta(r+1) = 0.5$ if 100 ms $< |CORR_{AEFTMA}^r| \leq 150$ ms, and $\beta(r+1) = 1$ if $|CORR_{AEFTMA}^r| > 150$ ms. Such a function was selected according to the type of load used during our experiments, which consisted of long-run applications that generate continuously high CPU load and network utilization.

Note that when $\beta(r) = 1$ the clock correction term resulting from AEFTMA equals that computed via the standard FTMA convergence function. This allows synchrony to be quickly reestablished in the presence of a transient fault, since the lagging effect of previous clock adjustments persists for at most one round after the fault occurred. After that, AEFTMA enters a memoryless operation mode, behaving exactly like FTMA, until $\beta(r)$ is again reduced below 1.

3.3: Sliding window algorithm

The sliding window algorithm (SWA) tolerates considerably higher percentages of *non-Byzantine* faults than other interactive convergence algorithms [3]. However, the algorithm requires that the number of Byzantine faults must be limited to $b \leq n/4$, which is a proper assumption for many systems. A basic assumption made by SWA is that the readings of different "good" clocks by a given node j should differ by a small amount. Therefore, it is expected that the clock estimates computed by j from the messages received from "good" clock sources will be clustered within a limited range. The basic operation performed by SWA consists of locating a cluster of "good" clock values by means of a sliding window mechanism. Alternatively, SWA can be implemented such that identification of clusters occurs in a clock deviation vector, which is the approach used in our implementation.

Assume that, at a given round r of the algorithm, a sorted clock deviation vector $\overline{X_j^r} = [x_1\ x_2\ \cdots\ x_i\ \cdots\ x_n]$, $x_i \leq x_{i+1}$, is available at node j (note that the elements in $\overline{X_j^r}$ are exactly the elements in $\overline{\Delta_j^r}$, except that in $\overline{X_j^r}$ they are sorted). Starting at the leftmost element of $\overline{X_j^r}$, SWA slides a window of width w to locate clusters of clock deviation values. This is done by aligning the left border of the window with each value $x_i \in \overline{X_j^r}$. The window spanning the range of values $[x_i, x_i + w]$ is referred to as the ith *window instance* (w_i). If $x_\ell \in \overline{X_j^r}$ is the largest clock deviation value such that $x_i \leq x_\ell \leq x_i + w$, then the *cardinality* of w_i (the number of clock deviation values in w_i) is given by $p_i = \ell - i + 1$. Among all n possible window instances, SWA selects a window instance w_e such that $\max(p_1, p_2, \ldots, p_n) = p_e$. If two or more window instances $w_a, w_b, \ldots$ exist such that $p_a = p_b = \ldots = p_e$, then an additional criterion is used for window selection. Two possibilities are to deterministically select the first window instance with cardinality p_e, or to select the window instance with minimum variance among those with cardinality p_e. Once a window instance w_e is selected, the clock correction term can be calculated by taking either the mean or the median of the values in w_e. This results in a total of four variations of the algorithm: SWA_{median}, SWA_{median}^{det}, SWA_{mean}, and SWA_{mean}^{det} [3]. Due to several advantages discussed in [3], SWA_{mean}^{det} was selected for our implementation. The clock correction term in node j ($CORR_j$) computed by SWA_{mean}^{det} is given by $CORR_j = (p_e)^{-1} \sum_{i=e}^{e+p_e-1} x_i$.

The size of the window frame (w) is an important parameter in all variations of SWA. For SWA_{mean}^{det}, the optimal window size is $\Delta_{int} + d_{var}$ [3]. A window of size $w = 100$ ms was selected for the implementation of SWA used during our experiments. This was done because about 70% of all "good" clock deviation estimates that were collected during our experiments with the SWA algorithm remain below the 100-ms threshold, meaning that a window of similar size correctly selects "good" clocks with very high probability.

4: Experimental results

4.1: Message delay distribution and algorithm performance

The experiments described in this section were conducted in an environment in which the message delay typically follows a distribution similar to that shown in Figure 1.[1] The main characteristic of interest is the long and thin tail of the distribution. Although a maximum delay d_{max} may be present, its value is usually much greater than d_{min}.

While the vast majority of message delays occur in a small range around the expected delay, outliers (messages with very long delays) do occur with some regularity. It is these

[1] A similar distribution was reported in a different environment by Cristian [6].

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.