SELF SYNCHRONIZING CLOCKS FOR REAL TIME SYSTEMS

Mohamed Benmaiza* and Murat Tayli†

College of Computer and Information Sciences, King Saud University
P.O. Box 51178, Riyadh 11543, Saudi Arabia

الخلاصـة :

إن الدوريْـن الرئيسييْـن اللذيْـن تقوم بهما خوارزميات تزامن عدادات الزمن هما: دقة مجموعة من العدادات بالنسبة بعضها لبعض، وضبطها بالنسبة لزمن مرجعي حقيقي. فالأولى تُبقي الفرق بين أيِّ عدادين من المجموعة ضمن حدود معروفة. أما الثانية، فالهدف منها هو إبقاء الفرق بين أي عداد داخل المجموعة والعداد المرجعي في حدود معينة. في حين أن النظم الموزعة غير المعتمدة على الزمن تحتاج إلى دقة العدادات فقط، فإن نظم الزمن الحقيقي لا يمكنها أن تستغني عن الضبط بالنسبة لزمن مرجعي حقيقي. إن العديد من الخوارزميات المبرمجة للقيام بتزامن العدادات تعتمد فقط على دقتها متعاملة مع مشكلة ضبط العدادات على أنها ثانوية مما يؤدي عادة إلى حلّها بصفة مكلّفة. مع التطوير المتزايد لنظم الزمن الحقيقي الموزعة فقد أصبح لازما تصميم طريقة تأخذ بعين الاعتبار – في نفس الوقت وبصفة متوازية – دقة وضبط العدادات. وقد قام هذا البحث بتصميم وتطبيق خوارزميات مبرمجة لتزامن العدادات تهدف إلى الحصول على دقة وضبط العدادات في آن واحد. إن الخوارزمي المقترح مبني على نمطية جديدة تُسمى [التزامن الذاتي]، والذي يتم من خلالها تزامن ذاتي مستمر من طرف أي عداد بالنسبة للعداد المرجعي، وبصفة مستقلة عن العدادات الأخرى. ويتم هذا بالقيام بأقل قسط ممكن من تبادل الرسالات (عبر شبكات الحاسوب). إن الجانب المهم لهذه النمطية هو إمكانية القيام بتطبيق خوارزميات تزامن العدادات بحيث تكون أقل تكلفة وقابليتها لتحمل الأخطاء دون الرجوع إلى أي أجهزة إلكترونية مختصة. وقد برهنا أخيرا على أنه باستعمال هيكلة مناسبة فإن مستوى الدقة الذي يمكن الحصول عليها تضاهي أيَّ حلٍّ مرتكز على الأجهزة الإلكترونية من نوع ال VLSI.

e. mail: *benmaiza@ccis.ksu.edu.sa
†murat@acm.org

ABSTRACT

The two functions achieved by clock synchronization algorithms are *clock precision* and *clock accuracy*. The former keeps the drift between any two clocks in a given set of clocks within defined limits; the latter maintains the drift between a given clock and a reference clock within defined limits. While distributed non real-time systems need only clock precision for their correct operation, clock accuracy is an absolute must for distributed real-time systems. Many of the existing software-based clock synchronization algorithms implement only clock precision, treating clock accuracy as a secondary problem, often solved at a high cost. As modern distributed real-time systems are emerging more and more, it becomes necessary to devise an approach that deals with the precision and accuracy issues equally and in an integrated way. The work presented in this paper consisted in the design and implementation of a software-based clock synchronization algorithm, which achieves at the same time clock accuracy and clock precision. The proposed algorithm is built upon a new mechanism, referred to as *self synchronization*, through which a clock can continuously synchronize itself relative to a reference clock, independently of the others and with minimal message exchange. An important aspect of this mechanism is to permit the implementation of a very low cost, fault-tolerant algorithm without resorting to any specific hardware. We finally show that, with proper architectural support, the level of attainable accuracy can match hardware-based solutions.

Index Terms: clock synchronization algorithm, clock accuracy, clock precision, distributed real-time systems.

SELF SYNCHRONIZING CLOCKS FOR HARD REAL TIME SYSTEMS

1. INTRODUCTION

Since the pioneering work by L. Lamport on event ordering in distributed systems [1], clock synchronization has been widely recognized as a fundamental issue for the correct operation of distributed systems: global synchronization, global update and recovery problems, maintenance of a global reference time for real-time systems, to cite a few, cannot be properly handled if clocks at different sites are not closely synchronized. As the abundant literature reveals [2–12], quite intensive research has been conducted on the subject since 1978, and many solutions, distributed or centralized, software and/or hardware-based have been proposed.

Clock synchronization encompasses two distinct issues: the clock precision within pre defined limits, and the clock accuracy relative to a known real time referential [4, 5]. In the general context of distributed systems, the clock synchronization problem has as objective the approximation of the global reference time, in presence of arbitrary communication delays and clock drifts. Estimate of the reference time is then used to build clock precision within defined limits. However, these approaches reveal to be insufficient for real time systems that must rely on clock accuracy for their correct operation. An important point to note is that clock accuracy cannot be achieved using only estimates of the reference time; it needs exact knowledge of the reference time.

Software-based clock synchronization algorithms implement, in general, techniques to achieve clock precision, and tackle the clock accuracy issue as an additional problem that can be solved at some extra cost, whenever required [4, 6]. Hardware solutions providing clock accuracy already exist, but are expensive [4]. We present in this paper a software-based alternative that builds clock accuracy and avoids the replication of hardware equipment that induces significant problems at system integration stage. As clock accuracy is a stronger condition than clock precision (in the sense that the accuracy of a set of clocks also involves their precision), this solution would de facto guarantee clock precision. The proposed approach assumes arbitrary clock drift rates as in other algorithms, but differs from them by relying on predictable communication delays. Although, the latter assumption seems restrictive, it is however a reasonable tenet for real time distributed systems for which the predictability of basic system services, including that of communication services, is an absolute must [13, 16].

A generic characteristic of software-based solutions is their reliance on collective decision making process to synchronize their clocks. A community of sites interact through heavy message exchanges to form a consensus about a common time base. Major drawbacks of this cooperative operation are: (*a*) a non-negligible increase in message traffic and inter-process communication costs; (*b*) an added inaccuracy resulting from clock reading errors at the different sites; and (*c*) the tendency to synchronize, or rather to lock, to the fastest clock in the community, drifting away significantly from the real time.

Piggybacking techniques, as suggested in [5], can be efficiently used to minimize the number of exchanged messages and reduce communication costs. The reliance on message exchanges for clock synchronization purpose is inevitably a source of clock reading errors because of the uncertainties in the transfer and processing times of these messages. This type of errors can be contained through a tight control of unpredictability factors induced by the communication subsystem and the operating system. However, an ultimate solution to issues (*a*) and (*b*), would consist in eliminating message exchanges for clock synchronization purposes altogether. Finally, locking to the fastest clock can be avoided by referring to an absolute source of time, rather than deriving it using some quorum or consensus protocols.

The solution proposed in this paper claims to address the problems stated above in an integrated way, using a scenario whereby each site in the system would synchronize itself with respect to a reference time source, and maintain its synchronization without having recourse to specific message exchanges. Because of the way it operates, the proposed solution is referred to as *self-synchronizing clocks.* The advantages of such an approach are manifold:

- very low cost as the number of messages exchanged is very low;
- fault-tolerance, because each site synchronizes itself in an isolated way and the reference site can be replicated; and
- applicability to both real-time and non real-time systems.

Last but not least, the proposed solution also claims that, with the proper architectural support, it is possible to achieve clock accuracy within the same order of magnitude as hardware-based solutions.

How the proposed clock synchronization algorithm precisely works is explained in Sections 3 and 4. Section 2 restates more formally the clock synchronization problem and gives basic definitions. Section 5 presents implementation problems in general, while Section 6 discusses the integration of the proposed algorithm in the kernel of the *Real Time Distributed Operating System* (RTDOS) [16]. Finally, the paper concludes by a summary of the main features and strong points of the self-synchronizing clocks approach.

2. PROBLEM STATEMENT

Clock synchronization is seen as a set of techniques (implemented by specific algorithms) to create and maintain a consistent global time base for a set of, possibly faulty, interconnected sites in a distributed system. Many clock synchronization techniques based on message exchanges have been developed. The *clock self-synchronization technique* presented in this paper differs from others in that it relies on minimal message exchange. Foundations of our approach and necessary definitions are introduced in this and the following sections.

Logical Clock

Each site S_i has knowledge of the global time through a local logical clock denoted C_i. C_i can be defined as a monotonic, increasing function where $C_i(t)$ is the reading of the value of logical clock C_i at real time t. It is clear that the value of C_i is the only observable time at the site S_i.

Physical Clock

A given site S_i is equipped with a physical (hardware) clock PC_i through which the logical clock C_i is derived and maintained by the clock synchronization process. Usually, the granularity of C_i is much larger than the granularity of PC_i. Since a physical clock is never accurate, its drift from a reference real time clock is non-null, implying that the drift rate of the derived logical clock is also non-null.

Clock Rate

The rate ρ of a physical or logical clock c, over an interval of real time $[t_1, t_2]$, is defined as:

$$\rho = \left| \frac{c(t_2) - c(t_1)}{t_2 - t_1} \right| .$$

A perfect clock c is such that $c(t) = t$, $\forall\, t$, giving $\rho = 1$ second/second, $\forall\, t$, if the time unit is the second.

Drift Rate

The drift rate δ of a clock c is the number of time units (or clock ticks) by which c deviates in one unit of real time. It can be formally defined as $\delta = |1-\rho|$. A non faulty clock c is such that $\delta < d$, for a given d. The limit d can be reasonably taken as $1 \gg d \geq 0$, which means that a non-faulty clock is considered as drifting by a small fraction of time unit per real time unit.

Clock Reading Error

Clock synchronization is usually based on message exchanges. A given site S_i has knowledge of the time of a given site S_j through a message $\mathcal{M}$, carrying the clock value of the site S_j. In order to "read" correctly j's clock, site S_i must account for transmission and processing times of the message $\mathcal{M}$. Since there are variations in message transmission and processing times at different sites, estimates for successive readings of site S_j's clock at site S_i vary as well. These alterations in clock reading times are called *clock reading errors* or *reading errors* in short.

Logical Clock Precision

For a defined period of real time T, a collection of n sites are said to have their logical clocks precise within the limit ε, if

$$\forall t \in T,\ \forall i,j \in \{1,2,\ldots,n\},\ |C_i(t) - C_j(t)| \leq \varepsilon.$$

A major objective of clock synchronization algorithms is to maintain logical clock precision within specified values of ε. This type of clock synchronization, called *internal synchronization* [4], is usually implemented by having all concerned sites periodically exchange messages containing some sort of clock information. The frequency of clock synchronization messages is defined according to a re-synchronization period π. Every π, a given site S_i adjusts its logical clock according to an agreed upon synchronization protocol, after the reception of the messages carrying clock information from all other sites. The synchronization protocol must be defined so that clocks can only be set forward in order to avoid having negative time intervals [4–6]. The consequences of this are that clocks tend to be adjusted to the fastest clock in the system on one hand, and on the other, global time tends to drift away importantly from the real time. This fact is quite acceptable if global time is solely used for event ordering. Yet, it may not be appropriate for real time systems that need to refer to an absolute time referential.

Mechanisms for internal clock synchronization fall in two categories: instantaneous re-synchronization and continuous re-synchronization. In the first case the correction is immediately applied every re-synchronization period; and in the latter, a clock is corrected by spreading the adjustment over the next re-synchronization period. Such a continuous adjustment is implemented through continuous clock rate correction [4].

Clock Accuracy

Call RTC a reference source of real time t (Temps Atomique International provider for example). A logical clock C_i is said to be accurate relatively to a real time clock RTC if

$$\forall t,\ |C_i(t) - t| \leq \theta,$$

where θ is an arbitrarily small positive value.

Clock accuracy is achieved by what is known as *external synchronization* which builds, at a given site S_i, a view of a reference real time clock [4]. Usually, such mechanisms come to derive the real time from the local logical clock, using specific mapping functions. As a consequence, the accuracy of a given logical clock depends highly on its precision.

Note that continuous clock correction applied to a logical clock tends in fact to put an upper bound on the drift rate of a logical clock, relative to other logical clocks. However, it does not prevent the drift from a reference real time clock [5]. If we were to ensure clock accuracy, then it becomes necessary to design mechanisms that allow to bound the absolute value of any logical clock, relative to a known source of real time. The use of continuous clock correction to adjust the rate of a logical clock, with respect to the rate of a reference real time clock, would help bound the absolute value of a logical clock.

As stressed in the introduction, clock accuracy is a stronger condition than precision. The following theorem announces formally this condition.

Theorem 1

For a set of n sites in a distributed system

$$\text{clock accuracy} \Rightarrow \text{clock precision}.$$

Proof

Take n sites $S_1, S_2, \ldots, S_n$, with accurate logical clock, *i.e.* for which

$$\forall t,\ \forall i \in [1,n],\ |C_i(t) - t| \leq \theta.$$

This involves that

$$\forall t,\ \forall i,j \in [1,n],\ |C_i(t) - C_j(t)| \leq 2\theta,$$

showing that the clocks are precise within 2θ.

□

3. SELF SYNCHRONIZATION PROCESS

A physical clock is built from an oscillating quartz crystal, and ticks at a given rate with a known accuracy. A logical clock is usually implemented as a *time counter*, derived from a physical clock by periodical increments, as illustrated in Figure 1. The counting period η depends on the desired time granularity of the logical clock.

The rate of a physical clock PC relative to the real time t, $\rho = dPC(t)/dt$, defines the accuracy of the physical clock, which in turn defines the accuracy of the derived logical clock. The equality $dPC(t)/dt = 1$ defines a perfect physical clock, $dPC(t)/dt > 1$ a fast physical clock, and $dPC(t)/dt < 1$ a slow physical clock. It is interesting to note that, if a site knows precisely the drift rate of its logical clock, relative to a real time reference, over a given time interval τ, then it can correct its logical clock C without resorting to message exchanges.

For a given logical clock C, with a drift rate δ, the self synchronization will be performed every $\pi = 1/\delta$, π defining the period where C is exactly one unit fast or slow relative to the real time clock. The correction algorithm will simply consist in dropping from, or adding to, C one extra unit, depending on whether C is fast or slow (Figure 1).

Note that the correction period π is a discrete value, yet its computation can generate a fractional part that will be rounded to the closest integer. The inevitable inaccuracy introduced by the rounding operation has also to be compensated for. It is crucial that this correction, too, should set a logical clock only forward in order to avoid negative intervals. Therefore, the rounding operation should always perpetrate a slow down of a logical clock, so that the readjustment would consist of setting the clock forward by a number of units corresponding to the possible deviation. Subsequently, rounding of π is subject to the following rules:

- if a logical clock is fast ($\rho > 1$), round down π so that we "drop" more frequently 1 unit, leading to a slow clock over the period τ,
- if the clock is slow ($\rho < 1$), round up π so that we "add" less frequently 1 unit, leading to a slow clock over the period τ.

Let us now estimate the deviation introduced by the rounding process. Call π' the rounded value of π ($1 > |\pi - \pi'| \geq 0$).

Lemma 1

The maximal deviation of a logical clock induced by the rounding process is equal to 1 time unit every π'^2 time units.

Proof

The rounding process introduces an inaccuracy of $|\pi - \pi'|/\pi$ over every period π' and so a maximal inaccuracy of $1/\pi$ units over the period π'. Consequently, the deviation of a logical clock is less or equal than 1 time unit every π'^2 time units. The maximal deviation is then as stated in Lemma 1.

□

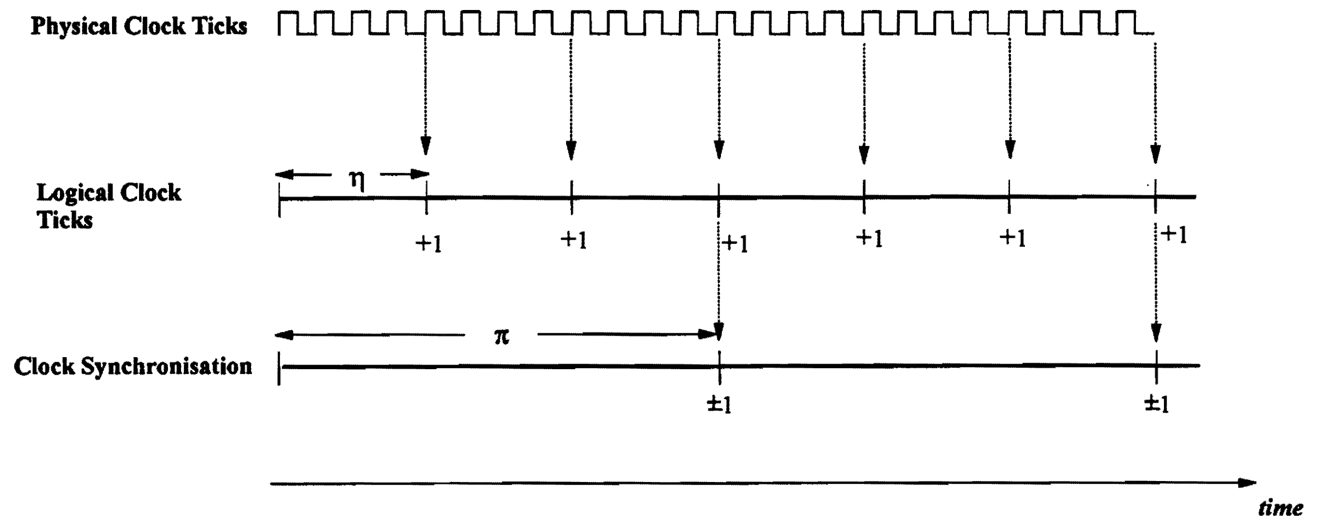

Figure 1. Counting and synchronization process.

Clock Readjustment

Lemma 1 involves that a logical clock must be readjusted by 1 time unit every π'^2 time units to compensate for the effect of the rounding process. Let us call π'^2 the *readjustment period*, and name it *R*. *R* defines, indeed, the time interval over which one extra unit should be *added* to the clock. If π' is large, as expected, then the inaccuracy will be significantly small, and the readjustment period very long. To fix the ideas about the importance of the rounding error and the readjustment period, let us consider a fast logical clock that deviates by +1 milliseconds (ms) every 10000.9ms. The self-synchronization period $\pi = 10000.9$ ms will be rounded to $\pi'=10000$ ms, causing an inaccuracy of 1 ms every 10^8 ms $\cong$ 28 hours.

Theorem 2

Given the drift rate ρ of a logical clock C_I and assuming that C_I has been properly initialized to some reference real time, the application of self-synchronization process (±1 unit) every period π' and readjustment process every period π'^2 guarantees the accuracy of the clock C_I.

Proof

The application of self-synchronization involves that (1) $dC_i(t)/dt \approx 1$ every π' and the application of readjustment involves that (2) $dC_i(t)/dt \approx 1$ every π'^2. (1) and (2) involve that $dC_i(t)/dt \approx 1$ $\forall t$ which guarantees the accuracy of C_i.
□

The self-synchronizing clocks approach does not necessarily assume the accessibility to a unique *Reference Time Provider* from every site in a distributed system. In fact, any site can act as an alternate *Reference Time Provider*, with a known accuracy that depends on the particular topology adopted. For example, Figure 2 depicts a possible hierarchy where the root of the tree at $level_0$ is the Universal Time Provider. $Level_i$ ($i > 0$) nodes are "rooted" to one node at $level_{i-1}$ that is also their Reference Time Provider. System architectures that accommodate multiple *Reference Time Providers* do not only simplify intricate problems that may arise during dynamic scaling of a distributed system, but they also provide high fault-tolerance *via* the replication of such providers across the system.

4. SELF SYNCHRONIZING CLOCKS ALGORITHM

Self synchronizing clocks algorithm consists of two distinct operational phases: (1) an initial *calibration phase* during which a node joining the system, at the start up or after a crash, adjusts its logical clock to a referential time and computes its drift; (2) a *self synchronization phase* during which the algorithm tries to keep different sites synchronized within defined limits.

The algorithm assumes that every concerned site in the system has access to a known provider of real time, RTP. However, the RTP can be any "trusted" node in the network: a node with a known deviation from Universal Time. A RTP node can be pre-calibrated (possibly manually) to serve as a time basis for the rest of the network nodes in the calibration process as explained below. Note that the RTP node uses the same self-synchronization algorithm as the other nodes to keep its clock as close as possible to Universal Time.

The *calibration phase* serves to determine the drift rate of site S_i's logical clock C_i, relative to RTP over a given period τ. The *self-synchronization period* π' and the readjustment period *R* are then derived from the drift rate. Based on the periods π' and *R*, clock correction procedure is applied according to Theorem 2 to build and maintain clock synchronization.

The calibration operation involves message exchanges between the site to calibrate and the source of real time. The necessary condition for the calibration to be performed correctly and drift rate to be determined precisely is to keep the *reading error* close to 0. This involves that factors contributing to the reading error, such as transmission delays, and message processing times, must be known precisely. Meeting the above condition is, to a large extent, a matter of implementation and will be discussed in Section 5. It is interesting to note that message exchanges required by the proposed approach take place only during the *calibration phase.* Therefore, in order to minimize reading errors, usage of "heavy" techniques may be considered at this stage, without major drawbacks on the cost or performance of the algorithm.

In the following, we adopt the convention to represent the real time by small letters (t, for example) and the logical time by dashed small letters (t', for example). Let τ be a period of real time known to every site involved in clock synchronization and τ'_i, the value of τ as seen by a site S_i. Clearly, τ'_i may be greater, equal, or less than τ depending on whether logical clock C_i is fast, exact, or slow relative to RTP. The two phases of the algorithm are given below.

Calibration phase

1. Get the initial time t_0 from closest RTP
2. Compute $t'_0 = t_0 + t_{rtc,i}$, where $t_{rtc,i}$ represents communication and processing overheads for the site S_i
3. Initialize logical clock C_i to t'_0, and start the counting process to increment C_i by one unit every η ticks of site S_i's physical clock
4. After τ'_i, get time t_1 from RTP and compute $t'_1 = t_1 + t_{rtc,i}$
5. Compute clock rate $\rho = |(t'_1 - t'_0)/(t_1 - t_0)|$ for the interval $\tau'_i = |t'_1 - t'_0|$
6. Derive the drift rate $\delta = |1 - \rho|$
7. Determine the self synchronization period $\pi' = \text{round}(1/\delta)$
8. Determine the readjustment period $R = \pi'^2$

end phase

Self synchronization phase

```
Every η do /* counting period */
begin
if π' units have elapsed /* self synchronization period */ then
    if ρ > 1 /* logical clock Ci is fast */ then
            do nothing; /* forget one tick */
    else    if ρ < 1 /* logical clock is slow */ then
                    Ci=Ci+2; /* catch up the time */
            else do nothing /* ρ = 1, perfect clock */
    else    Ci=Ci+1; /* normal clock update */
end;
Every R units /* readjustment period */ do
    Ci=Ci+1;
```

end phase

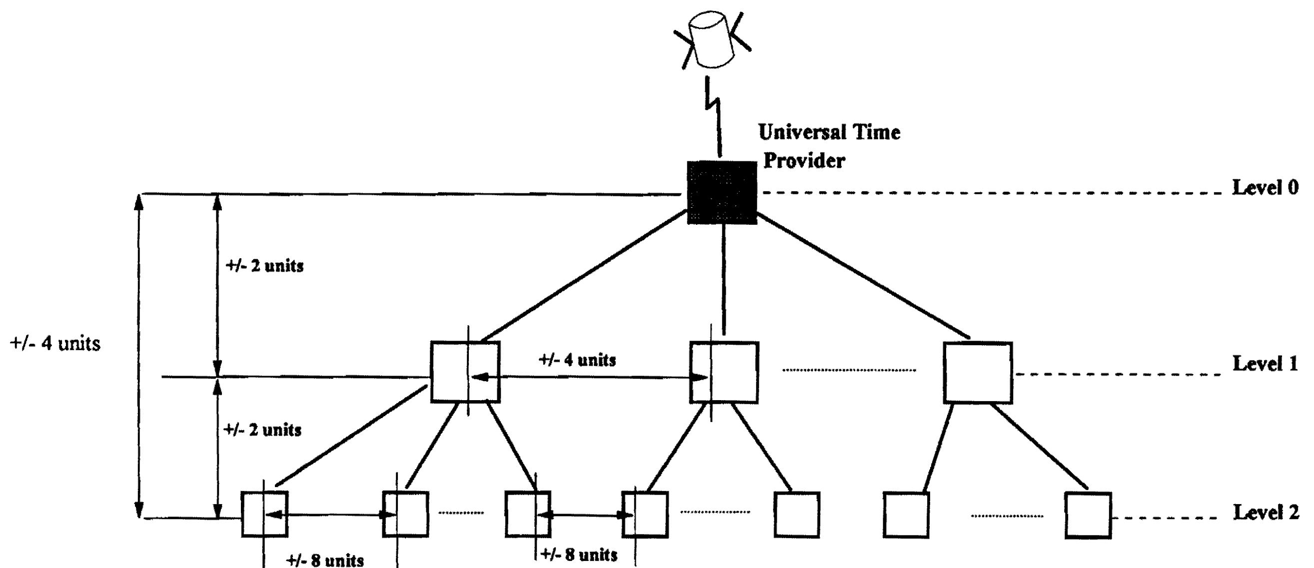

Figure 2. Time Deviation in a hierarchical tree-structure.

Note that the algorithm assumes the uniformity of time units between RTP and a given site S_i. If this is not the case, then a mapping function can be applied to align all units in a consistent way. It is also implicitly assumed that drift rates of various physical clocks are deterministic and stable within defined operational conditions (especially range of temperatures within which clock operation is stable). If a drift rate is not stable, the physical clock is considered as faulty. However, re-calibration at defined intervals can be envisaged as a possible way to compute again the drift rate and keep a tighter clock synchronization, when clocks with unstable drift rates are considered to be non faulty.

5. GENERAL IMPLEMENTATION ISSUES

The *Self-Synchronizing Clocks* algorithm, presented in this paper, is largely independent from the architecture of the experimental RTDOS platform. Its implementation does not require the provision of special hardware or software components. As such, it can be adopted for a wide range of distributed systems, extending from large heterogeneous general purpose systems to embedded real time systems, provided they rely on predictable communication delays and handle properly implementation pitfalls. Data networks implementing point to point communication (*i.e.* through circuit switching as in ISDN [17]) are appropriate platforms to build predictable communication services, and hence have the potential to allow an efficient implementation of the proposed approach.

There are several architectural constraints that may breach proper implementation of the clock synchronization process in a distributed system. It is to note that, most of these constraints are not just proper to the proposed approach, but they are also affiliated with all software based solutions that aim high clock precision. Correct performance of *Self-Synchronizing Clocks* algorithm rests on the dependability of two time-critical factors, both controlled by the underlying system: the accuracy of the reference time acquired for clock initialization and calibration activities, and the timeliness of counting and clock synchronization operations. It is also implicitly assumed that, concurrent system activities, if any, would not hinder significantly the timing of the overall process.

Given the cost and physical constraints, it is unlikely that every site, in a distributed system, can be equipped with a local provider of the reference time. Instead, a few servers scattered over the network will be in charge to provide the *Reference Time Service*. As the accuracy and the precision of the local time depend closely on the quality of the *Reference Time Service*, it is vital to analyze and evaluate major factors involved in the time acquisition process. Figure 3 sketches the scenario between a client process and a *Reference Time Server*. Major events affecting the course of actions at both sites are depicted in their chronological sequence mapped onto an unscaled time axis. It is to note that, local time at the client site will be set to the time sent by the server, offset by the acquisition cost σ_e, which corresponds to the time elapsed between the acquisition of the reference time by the server (T_{read}) and its delivery to the requesting process (T_{post}). In many cases, the cost of local processing may also be important to consider. The overall cost can then be defined as the quantity σ_t, corresponding to the interval $T_{update}-T_{read}$. It is composed of communication costs (t_2), process dispatching delays (d_2), and computation overheads (p_2 and p_3). The reading error corresponds, therefore, to the variations of the overall acquisition cost σ_t.

In many systems, the acquisition of the reference time by the time server (T_{read}) is not an event that can be observed from client sites. Therefore, a direct measure of time acquisition cost σ_t is not possible. A number of implementations resort then to the indirect estimation of σ_t, such as measuring the time elapsed from the initiation of the request to the receipt of the reply ($T_{post} - T_{request}$), and guessing the moment when T_{read} could have happened. Yet, this approach may not be appropriate for real time systems, as the nature of system components involved in the process prevents a deterministic evaluation of σ_t. It is to note that, IPC operations constitute a first source of reading errors. The use of shared communication media and the involvement of intermediary communication agents when server and client sites are not directly connected are among major uncertainty factors. Moreover, the cost and the occurrence of process dispatching activities cannot be anticipated and controlled readily. Finally, client and server processes may be interrupted during the processing of the service request at both sites. Consequently, any architectural support capable of removing, if not containing, these unpredictability factors constitutes a major asset to achieve higher accuracy.

Proper functioning of the clock synchronization algorithm might also be threatened if the implementation disregards the asynchronous nature of the ticking process of a physical clock and the counting process of a logical clock. In other words, clocks synchronization algorithm assumes that: (*a*) logical clocks are instantaneously incremented following the expiration of the interval counted in physical clock ticks; (*b*) the interval counter of the physical clock is regenerated independently from the handling of the logical clock. Given the chain of physical and logical events that may occur in a system, the probability of deferring the activation of the process managing logical clocks, by one or more physical clock ticks, is not

equal to zero. The counting process should then compensate such delays in order to avoid their repercussion on the next counting period.

6. RTDOS IMPLEMENTATION OF SELF SYNCHRONIZING CLOCKS

The *Real Time Distributed Operating System* - RTDOS project [16] has been initiated to investigate, and to experiment with design alternatives and implementation techniques, leading to a reliable and predictable distributed platform for hard real time applications. Clock synchronization process is implemented as part of RTDOS kernel. Accurate global time is available to all RTDOS processes to address the needs of real time applications, as well as to support the implementation of system functions that enforce system reliability and fault tolerance.

This section starts by introducing major features that shaped the overall implementation and contributed to the accomplishment of a level of accuracy within the same order of magnitude as hardware-based solutions. Implementation decisions, proper to RTDOS platform, are presented under the title *RTDOS Clock Synchronization Process.* The realization of counting, synchronization and readjustment operations is intentionally omitted, as their implementation is trivial when concerns expressed in Section 5 are properly addressed. The design and implementation of RTDOS *Reference Time Service* is particularly emphasized, as the accuracy of the overall process depends closely on the quality of this service.

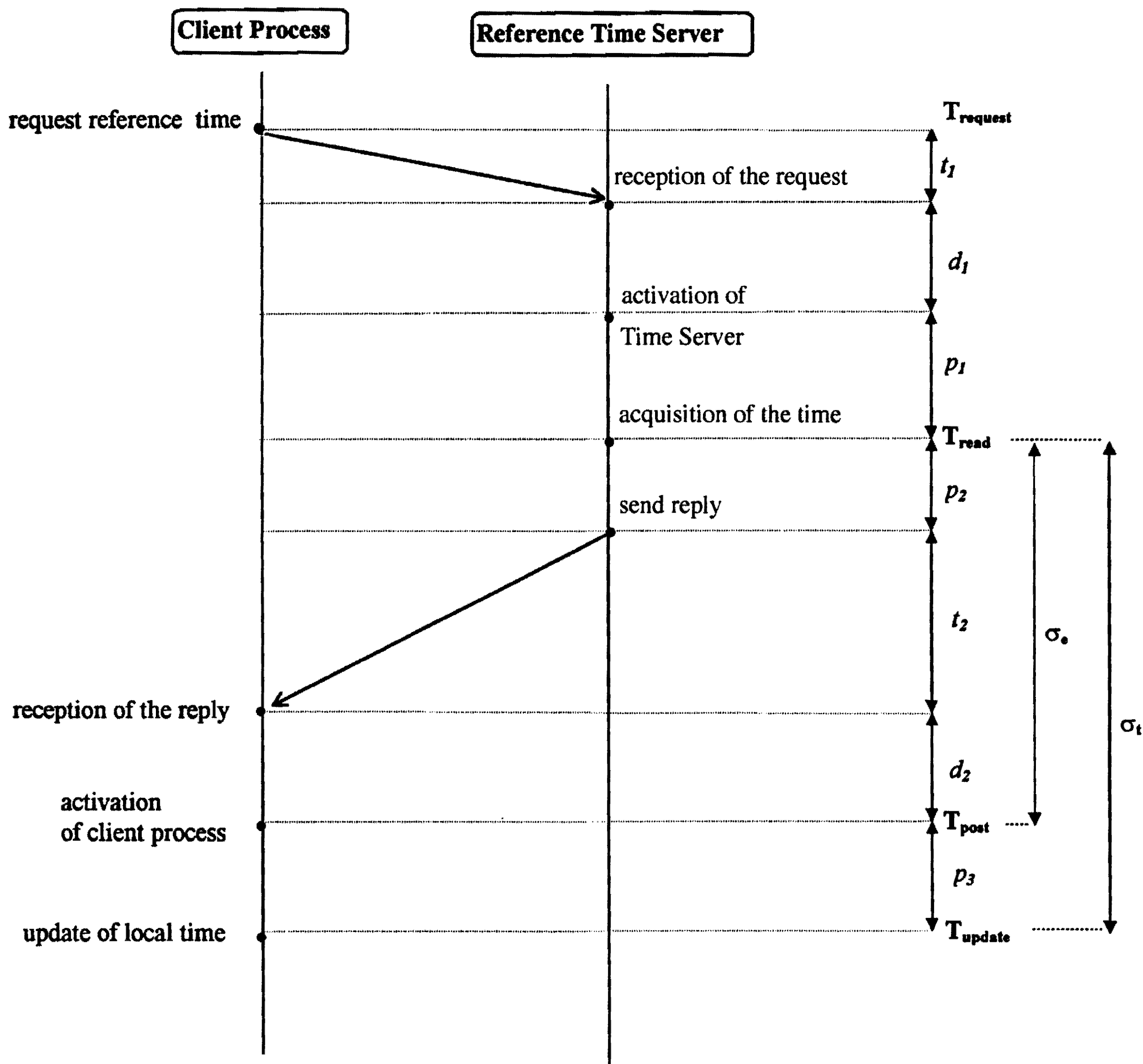

Figure 3. Analysis of time acquisition process.

Overview of RTDOS Architecture

RTDOS is designed to run on transputer networks organized as interconnected loops, or domains (Figure 4). Connecting two of the four serial communication links of a transputer to neighboring machines forms a given domain. The resulting bi-directional loop is dedicated to the usage of a limited number of kernel functions that communicate *via* an unreliable datagram service. The basic RTDOS configuration is a single loop referred to as the *base domain*. This base domain can be expanded to multi-domain configurations by connecting additional loops, *via* dedicated transputers called *Domain Managers*. RTDOS architecture does not impose *a priori* limits on the number of domains forming a system, nor the number of transputers in a given domain. RTDOS kernel, replicated in each node, provides necessary transparency to hide the underlying topology from the rest of the system. Functional partitioning of the system is left to the discretion of the projected application. Hardware and software components of a given system are configured during system generation phase, taking into account resource requirements, performance and reliability constraints of targeted applications.

RTDOS adopted the synchronous CSP communication model [3] as the basis of its inter-process communication (IPC) services. Furthermore, RTDOS–IPC has been extended to encompass N:1 relationship, a prerequisite to implement client–server paradigm for dynamically changing populations of processes. RTDOS processes, distributed within the system, communicate over non-shared, bi-directional, unbuffered, half duplex channels that provide total location transparency. RTDOS channels are mapped onto memory words when processes are located on the same transputer, and on physical circuits established with transputer links, when they are placed at different sites. A reliable connection service [15] establishes point-to-point connections, using two remaining transputer links that are wired either to programmable switches, or to fixed partners (Figure 4). Experimental results showed that, a single connection server was able to establish over 600 connections per second, with a guaranteed setup time less than 4 ms [14]. In general, RTDOS channels are mapped onto physical circuits, on request, and for the duration of a single exchange. However, time critical services, such as the *Reference Time Service*, may exceptionally settle long lived circuits, and avoid circuits reinstatement overheads for each exchange.

RTDOS Clock Synchronization Process

RTDOS implementation adapted the basic synchronization algorithm to meet its design objectives and to cope with its operational requirements. First, calibration of physical clocks has been identified as an off-line activity, to be undertaken

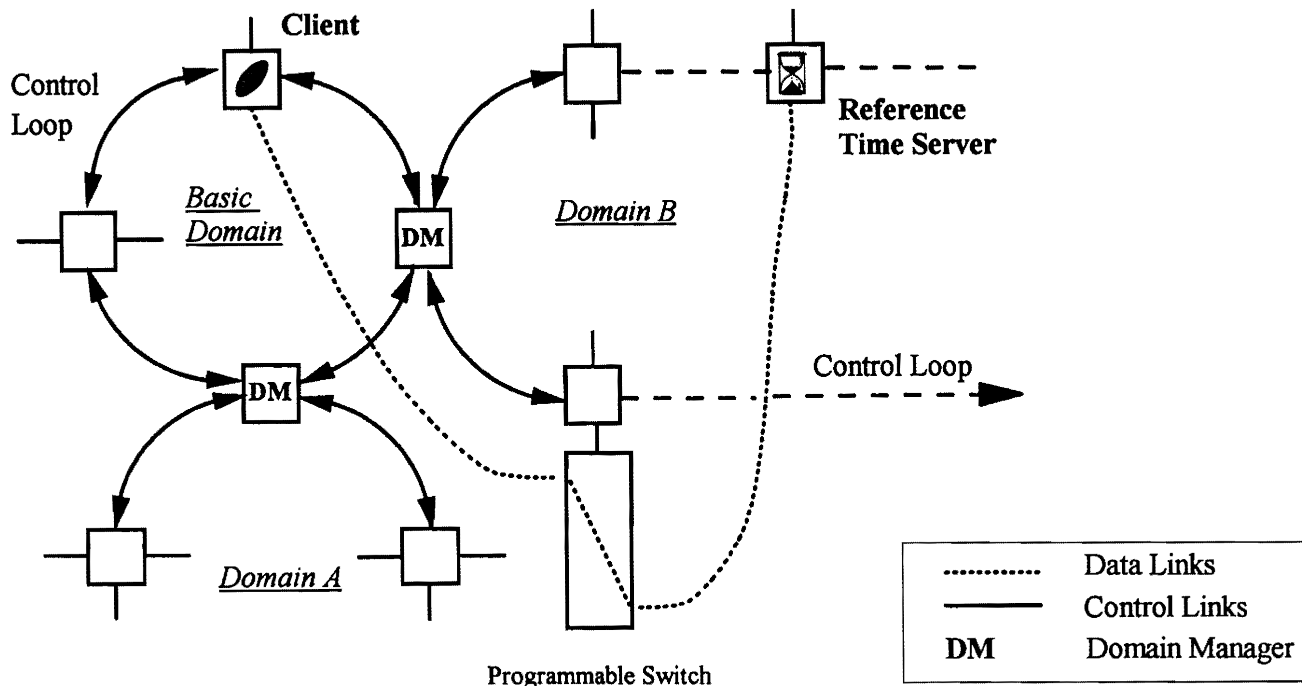

Figure 4. RTDOS architecture.

prior to the integration of transputers in a system. Secondly, a time auditing function has been introduced to monitor the performance of the time keeping process and to enforce the reliability.

The decision to calibrate physical clocks off-line stems from both physical and engineering considerations. Transputer hardware does not include internal physical clocks. Therefore, calibration activity concerns physical clocks rather then the transputer hardware. In many instances, a single external clock is used to drive a group of transputers. It is obvious that in such a context, calibration of individual transputer systems is a meaningless and redundant operation. Moreover, transputers cannot be physically inserted into, or removed from operational platforms. Thus, physical components of a given system must be integrated and tested in advance. It is to note that, hardware configuration of transputer based systems is static, but their topology may be dynamic.

During system start-up, RTDOS kernel resorts to a limited calibration process to assert the validity of the configuration data. System initialization proceeds with the alignment of the local time to the reference time and by the launch of two time-keeping processes: the *Time Manager* and the *Time Auditor*. The *Time Manager*, in charge of handling local logical clocks, performs the counting and self-synchronization activities specified by the basic algorithm. The *Time Auditor* monitors the performance of both physical and logical clocks. It first controls proper functioning of the physical clock driving the transputer, by performing a perpetual calibration. The drift rate of the clock is measured over a long period, and the findings are checked *versus* configuration data. A given system is declared faulty, when sizable variations in the drift rate are detected. As a concurrent activity, the *Time Auditor* also tracks possible shifts of the logical clock C_i, which may result from unaccounted system overheads interfering with counting periods. Deviations from the reference time, if any, are gradually offset by tuning the readjustment period R.

Organization of the Reference Time Service

Self-synchronizing clocks process relies on the *Reference Time Service* to initialize logical clocks, and to calibrate and track the performance of physical clocks within the system. The *Reference Time Service* can be implemented using a number of *Reference Time Servers,* organized in any appropriate structure such as the hierarchical tree structure presented in Section 3. It is the quality of this service that determines the accuracy of the overall process. Current implementation of RTDOS *Reference Time Service* guarantees an accuracy level of 10 µs by capitalizing on the possibility to observe remote events with accuracy, and the ability to confine reading errors. In comparison, note that the reading error has been evaluated to be 9 µs in the hardware based clock synchronization mechanism presented in [4] which involves that clock accuracy in this case can be 9 µs at best. This shows that our approach allows indeed to reach a level of accuracy within the same order of magnitude as hardware-based solutions.

RTDOS processes distributed within the system can coordinate their actions by means of synchronous IPC services. Figure 5 presents the protocol followed by *a Reference Time Server* and a client process for the acquisition of the reference time. First, *Reference Time Server* and client process establish a connection using the pair of primitives: *ConnectTo(TimeServer)* and *AcceptConnection(Client).* The opened connection will be exclusively used during the time acquisition operation and explicitly closed by the two parties. To reduce reading errors, client and server processes must act in a totally synchronized way during the time acquisition operation. For this purpose, an initial synchronization is achieved by the pair *Write(Synch) – Read(Synch).* We recall that in CSP-based communication, read and write operation are totally synchronous. After this synchronization point, time acquisition operation can be safely engaged (*Write(Reference Time) – Read(Reference Time)* pair).

As stated in Section 5, reading errors result mainly from uncertainties introduced by IPC services, process dispatching and unaccounted computation overheads. The design of RTDOS–IPC eliminates this major source of uncertainty, by providing predictable services. RTDOS–IPC communication costs t_1 and t_2 (Figure 3) are deterministic, since data is directly transferred from the source to the destination address space, over pre allocated media and communication hardware, without the interference of other concurrent activities. The second uncertainty factor, variations in processing times, is discarded by running the time service protocol at the highest priority. Client and server processes relinquish the processor control only voluntarily, *e.g.* for synchronous read–write operations. Therefore, processing times p_1, p_2, and p_3 (Figure 3) are also deterministic, as they are not subject to unexpected preemption.

Despite the under microsecond performance of context switching operation in transputer systems, process dispatching still remains the potential source of uncertainty. A high priority process that becomes ready to run, on occurrence of an event

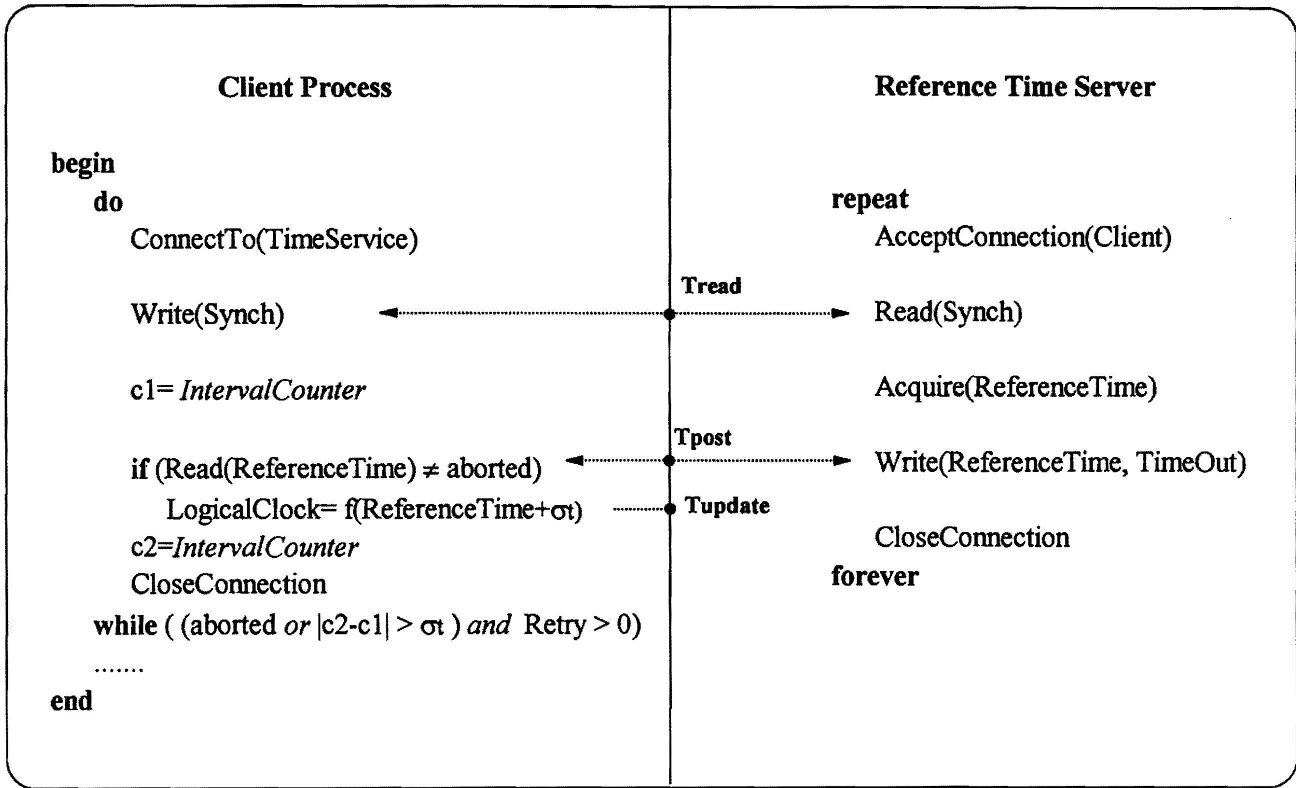

Figure 5. Reference time service protocol.

such as the end of a synchronous read or write operation, may not be able to regain immediately control of the CPU, due to the presence of other processes with similar precedence. As such incidents cannot be avoided *a priori*, time service protocol resorts to an indirect validation. It checks by means of an asymmetric timing test, whether a client or server process has been delayed at the dispatching stage. The *Reference Time Server* uses a write operation with a time out, to verify that the client has reached in time the corresponding read operation. In case a client is delayed after the synchronization point, the server deadline expires and the write operation is aborted. The client is notified when it reaches the read operation. Remaining deferment possibilities are: the suspension of the server after the synchronization point, and the postponement of the client activation following the receipt of the reference time. Both cases are detected by measuring the duration $c2-c1$ at the client site (Figure 5). Time acquisition operation is considered successful, if $c2-c1$ does not exceed the estimated time acquisition cost σ_t. In case one or the other system experiences unexpected delays, the transaction is considered void and reiterated until reference time is read within the specified time interval. On our experimental platform, the time acquisition cost σ_t is fixed at 50 µs. 40 µs correspond to the estimated processing cost, and 10 µs to reading errors. It is presumed that the 10 µs of reading errors are due to the variation in clocks speed among the sites and to the differences in their memory addressing architecture.

7. CONCLUSION

A new clock synchronization algorithm, adapted to the needs of both real-time and non real-time applications, has been presented in this paper. The major tenet of the proposed approach has been the consideration of the accuracy as the basic feature of any logical clock, clock precision being implicitly implied. The accuracy, taken as intrinsic feature of any logical clock, also conferred some of the forceful points of the approach: accuracy and precision are treated in a unified way; locking to the fastest clock is implicitly prevented; any site can be a provider of the *Reference Time Service* and temporarily substitute the original source.

The self-synchronizing clocks approach is highly fault-tolerant as synchronization decisions are taken by every site in total isolation, and reference time providers can easily be substituted. The autonomous character of synchronization process

contains the propagation of local faults, if any, and preserves the integrity of the sites in case of the failure of the original reference. Moreover, the accuracy of the local time at a given site offers the distributed system the possibility to replicate time servers and provide the *Reference Time Service* transparently.

The *Self Synchronizing Clocks* approach provides a low cost solution, in terms of inter-site communication traffic, and is largely independent of the implementation platform. The only implementation imperative is the predictability of the reference time acquisition process, used during the calibration of physical clocks and initialization of logical clocks. As the acquisition of the reference time is not a frequent operation, the use of complex and heavy mechanisms can be exceptionally tolerated to guarantee the required predictability. The algorithm has been successfully implemented on a transputer based platform, and preliminary measurements have shown an accuracy comparable to hardware-based clock synchronization algorithms.

REFERENCES

[1] L. Lamport, "Time, Clocks and the Ordering of Events in Distributed Systems", *CACM*, **21(7)** (1978), pp. 558–565.

[2] B. Liskov, "Practical Uses of Synchronized Clocks in Distributed Systems", *Distributed Computing*, **6** (1993), pp. 211–219.

[3] C.A.R. Hoare, "Communicating Sequential Processes", *CACM*, **21(8)** (1978), pp. 666–677.

[4] H. Kopetz and W. Ochsenreiter, "Clock Synchronization in Distributed Real Time Systems", *IEEE Transaction on Computers*, **C-36(8)** (1987), pp. 933–940.

[5] R. Drummond and O. Babaoglu, "Low-Cost Clock Synchronization", *Distributed Computing*, **6** (1993), pp. 193–203.

[6] P. Ramanathan, K.G. Shin, and R.W. Butler, "Fault-Tolerant Clock Synchronization in Distributed Systems", *IEEE Computer Magazine*, **23(10)** (1990), pp. 33–42.

[7] W.A. Vervoort *et al.*, "Distributed Time-Management in Transputer Networks", *Proceedings of the EUROMICRO'91 Workshop on Real-Time Systems*, 1991, pp. 224–230.

[8] M.J. Pfluegel and D.M. Blough, "Evaluation of a New Algorithm for Fault-Tolerant Clock Synchronization", *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems: Kawazaki, Japan*, September 26–27, 1991, pp. 38–43.

[9] J. Hue, Z. Mammeri, and J.P. Thomesse, "Clock Synchronization in Real-Time Distributed Systems based on FIP Field Bus", *Proceedings of the 2nd IEEE Workshop on Future Trends of Distributed Computer Systems, Cairo, Egypt*, 1990, pp. 135–141.

[10] S. Rangarajan and S. K. Tripathi, "Efficient Synchronization of Clocks in a Distributed System", *Proceedings of the 12th Real-Time Systems Symposium: San Antonio, Texas, USA*, December 4–6, 1991, pp. 22–31.

[11] C. Flaviu, "Probabilistic Clock Synchronization", *Distributed Computing*, **3** (1989), pp. 146–158.

[12] M. Raynal, "About Logical Clocks for Distributed Systems", *Research Report n° 1534, INRIA, France*, October 1991.

[13] J. Stankovic and K. Ramamritham, "The Spring Kernel, A New Paradigm for Real-Time Operating Systems", *ACM Operating Systems Review*, **23(3)** (1989), pp. 54–71.

[14] M. Benmaiza and M. Tayli, "Circuit-Switched IPC for Predictable Message Passing in a Multiloop Transputer Network", *Proceedings, World Transputer Congress, Aachen*, September 20–22, 1993, pp. 890–898.

[15] M. Tayli and M. Benmaiza, "An Efficient Circuit-Switching Mechanism for Inter Process Communication in a Transputer Network", *Proceedings, 4th IEEE Workshop on Future Trends on Distributed Computing Systems, Lisbon, Portugal*, September 22–24, 1993, pp. 215–220.

[16] M. Tayli, M. Benmaiza, and R. Eskicioglu, "RT-DOS A Real-Time Distributed Operating System Kernel for Transputers", in *Proceedings of the OUG 13, 13th OCCAM User Group Technical Meeting on Real-Time Systems with Transputers*, September 18–20, 1990, pp. 1–11.

[17] H. J. Helgert, *ISDN: Architectures, Protocols, Standards*. Amsterdam: Addison-Wesley, 1991 (ISBN 0-201-52501-1).

Paper Received 25 June 1997; Revised 1 September 1998; Accepted 19 October 1998.