HAL
archives-ouvertes.fr

Performance Evaluation of Clock Synchronization Algorithms

Emmanuelle Anceaume, Isabelle Puaut

▶ **To cite this version:**

Emmanuelle Anceaume, Isabelle Puaut. Performance Evaluation of Clock Synchronization Algorithms. [Research Report] RR-3526, INRIA. 1998. <inria-00073159>

HAL Id: inria-00073159
https://hal.inria.fr/inria-00073159

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Performance Evaluation of Clock Synchronization Algorithms

Emmanuelle Anceaume and Isabelle Puaut

N ° 3526

Octobre 1998

THÈME 1

ISSN 0249-6399

Performance Evaluation of Clock Synchronization Algorithms

Emmanuelle Anceaume and Isabelle Puaut

Thème 1 — Réseaux et systèmes
Projet Solidor

Rapport de recherche n3526 — Octobre 1998 — 36 pages

Abstract: Clock synchronization algorithms ensure that physically dispersed processors have a common knowledge of time. This report proposes a survey of software fault-tolerant clock synchronization algorithms: deterministic, probabilistic and statistical ; internal and external ; and resilient from crash to Byzantine failures. Our survey is based on a classification of clock synchronization algorithms (according to their internal structure and to three orthogonal and independent basic building blocks we have identified), and on a performance evaluation of algorithms constructed from these building blocks. The performance evaluation is achieved through the simulation of a panel of fault-tolerant clock synchronization algorithms [LL88, ST87, PB95, GZ89]. The algorithms behavior is analyzed in the presence of various kinds of failures (crash, omission, timing, performance, Byzantine), both when the number and type of failures respect the fault assumptions made by the algorithm and when fault assumptions are exceeded. Our survey will help the designer in choosing the most appropriate structure of algorithm and the best building blocks suited to his/her hardware architecture, failure model, quality of synchronized clocks and message cost induced. Moreover, our classification uses a uniform notation that allows to compare existing clock synchronization algorithms with respect to their fault model, the building blocks they use, the properties they ensure and their cost in terms of message exchanges.

Key-words: Clock synchronization, simulation, performance evaluation, distributed systems, real-time systems

(Résumé : tsvp)

This work is partially supported by the French Department of Defense (DGA/DSP), #96.34.106.00.470.75.65.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Evaluation de performance d'algorithmes de synchronisation d'horloges

Résumé : Les algorithmes de synchronisation d'horloges offrent une notion commune du temps à des processeurs n'ayant pas accès à une horloge globale partagée. Ce rapport propose un état de l'art des algorithmes de synchronisation d'horloges implantés par logiciel, qu'ils soient déterministes, probabilistes ou statistiques, qu'ils assurent une synchronisation interne ou externe, et qu'ils soient résilients aux défaillances par arrêt, par omission ou Byzantines. Cet état de l'art repose sur une classification de ces algorithmes et sur une évaluation de leur performance effectuée à partir de cette classification. Cette évaluation de performance est obtenue par simulation d'une sélection d'algorithmes de synchronisation d'horloges [LL88, ST87, PB95, GZ89]. Le comportement de ces algorithmes est analysé en présence de différents types de fautes (arrêt, omission, performance, Byzantines), à la fois lorsque le nombre et le type des fautes générées lors de l'évaluation respectent les hypothèses faites par les algorithmes et également lorsque ces hypothèses sont violées. Cet état de l'art est destiné à guider le concepteur d'un algorithme de synchronisation d'horloges dans le choix de la technique la mieux adaptée à son architecture matérielle, le modèle de défaillances visé, la qualité de la synchronisation obtenue ainsi que le coût résultant en terme de nombre de messages échangés.

Mots-clé : Synchronisation d'horloges, simulation, évaluation de performance, systèmes distribués, systèmes temps-réel

1 Introduction

A distributed system consists of a set of processors that communicate through message exchanges and do not have access to a global clock. Nonetheless, an increasing number of distributed applications, such as process control applications, transaction processing applications, or communication protocols, require that synchronized clocks be available to have approximately the same view of time. Time in this context means either an approximation of real time or simply an integer-valued counter. The algorithms ensuring that physically dispersed processors have a common knowledge of time are called *clock synchronization algorithms*.

Designing clock synchronization algorithms presents a number of difficulties. First, due to variations of transmission delays each process cannot have an instantaneous global view of every remote clock value. Second, even if all clocks could be started at the same real time, they would not remain synchronized because of drifting rates. Indeed clocks run at a rate that can differ from real time by 10^{-5} seconds per second and thus can drift apart by one second per day. In addition, their drift rate can change due to temperature variations or aging. The difference between two hardware clocks can thus change as time passes. Finally, the most important difficulty is to support faulty elements, which are a common fact in distributed system.

Clock synchronization algorithms can be used to synchronize clocks with respect to an external time reference and/or to synchronize clocks among themselves. In the first case, called *external* clock synchronization, a clock source shows real time and the goal for all clocks is to be as close to this time source as possible. Access to an external time reference is usually useful in loosely coupled networks or real-time systems. For instance, the NTP protocol [Mil91] is used for external synchronization on the Internet. In the second case, called *internal* clock synchronization, real time is not available from within the system, and the goal is then to minimize the maximum difference between any two clocks. An internal clock synchronization algorithm enables a process to measure the duration of distributed activities that start on one processor and terminate on another one. It establishes an order between distributed events in a manner that closely approximates their real time precedence.

Clock synchronization may be achieved either by *hardware* or by *software*. Hardware clock synchronization [SR87, KO87] achieves very tight synchronization, through the use of special synchronization hardware at each processor, and uses a separate network solely for clock signals. Software clock synchronization algorithms [CAS86, HSSD84, ST87, LL88, LMS85, VCR97, PB95, GZ89, MS85, FC95a, FC97, MO83, OS94, Arv94, Cri89] use standard communication networks and send synchronization messages to get the clocks synchronized. They do not require specific hardware, but do not provide synchronization as tight as hardware algorithms. Software clock synchronization algorithms decompose themselves in *deterministic*, *probabilistic* and *statistical* algorithms. Deterministic algorithms assume

that an upper bound on transmission delays exists. They guarantee an upper bound on the difference between any two clock values (*precision*). *Probabilistic* and *statistical* clock synchronization algorithms do not make any assumptions on maximum message delays. While probabilistic algorithms do not assume anything about delay distributions, statistical algorithms assume that the expectation and standard deviation of the delay distributions are known. Probabilistic algorithms guarantee a constant maximum deviation between synchronized clocks, while statistical algorithms do not. In probabilistic algorithms, a clock knows at any time if it is synchronized or not with the others, but there is a non-zero probability that a clock will get out of synchronization when too many unmasked communication failures occur. In statistical algorithms, clocks do not know how far apart they are from each others, but a statistical argument is made that at any time, any two clocks are within some constant maximum deviation with a certain probability. Note that we are not concerned in this paper with algorithms aimed at maintaining logical clocks that reflect properties on the execution of distributed applications, such as causal relationships [Lam78]. In addition, we do not detail issues specific to clock synchronization in large scale distributed systems (e.g. [SR87, VCR97]).

Clock synchronization has been extensively studied for the last twenty years. Thorough surveys can be found in [Sch86, RSB90] and [SWL90]. In [RSB90], software and hardware clock synchronization algorithms are classified with regard to the clock correction scheme used. In contrast, the algorithms surveyed in [SWL90] are listed according to the supported faults and the system synchrony (knowledge of upper bounds on communication latencies). Schneider's work [Sch86] gives a single unifying paradigm and correctness proof that can be used to understand mostly all deterministic clock synchronization algorithms supporting Byzantine failures.

Our work extends these surveys by proposing a taxonomy adapted to all published software clock synchronization algorithms: deterministic, probabilistic and statistical, internal and external, and resilient from crash to Byzantine failures. Our taxonomy is more precise than existing classifications in the sense that algorithms are classified according to a larger set of parameters: system synchrony, failure model, structure of the algorithm and three basic building blocks we have identified. The first building block, called the *resynchronization event detection* block, aims at detecting the instant at which processors have to resynchronize their clock. The second one, called the *remote clock estimation* block, estimates the values of remote clocks. The third one, called the *clock correction* block, is designed to correct each logical clock according to the result provided by the remote clock estimation block. This taxonomy makes possible the design of any clock synchronization in terms of the three identified building blocks, which allows to compare the behavior of all clock synchronization algorithms in a common framework. In addition, we use throughout the paper a uniform notation that allows the comparison between all algorithms with respect to the assumptions

they make, the building blocks they use, the properties they ensure (e.g. precision) and their cost in terms of number of messages exchanged.

In addition to the classification of clock synchronization algorithms, our survey includes a performance evaluation of a panel of deterministic clock synchronization algorithms [LL88, ST87, PB95, GZ89] constructed from the building blocks identified in the taxonomy. The performance evaluation is achieved through simulation. Its object is twofold. It allows to compare the algorithms *tightness* (i.e. maximum difference between any two clocks) with their (worst-case) guaranteed precision, and allows to compare the algorithms behavior in the presence of failures. An analysis of the algorithms behavior is performed in the presence of various kinds of failures (crash, omission, timing, performance, and Byzantine), both when the number and type of failures respect the fault assumptions made by the algorithm and when fault assumptions are exceeded.

Our survey should help the designer in choosing the most appropriate structure of algorithm and the best building blocks suited to his/her hardware architecture, failure model, quality of synchronized clocks and message cost induced. For instance, our performance evaluation shows that properties of some building blocks allows to support more failures than expected with minor perturbations on the algorithm behavior.

The remainder of this paper is organized as follows. Section 2 describes the underlying system upon which clock synchronization algorithms are based. A precise statement of the problem to be solved is given in Section 3. Section 4 introduces our classification of clock synchronization algorithms, and illustrates it on a large set of algorithms. Section 5 is devoted to an experimental analysis of the behavior of a panel of clock synchronization algorithms in the presence of failures. Finally, we conclude in Section 6. A glossary is given in Section 7.

2 System model

The study of any clock synchronization algorithm requires an accurate description of the underlying distributed system, concerning its overall structure (section 2.1), the timing assumptions of its network (section 2.2) and the failure model of its components (section 2.3).

2.1 Processors, network and clocks

We consider a set $\mathcal{P} = \{p_1, p_2, ..., p_n\}$ of processors interconnected by a communication network. According to the algorithms, the network can have different characteristics (broadcast or point-to-point, fully-connected or not). Unless explicitly stated, a *fully connected* network is assumed throughout this paper.

With each processor $p_i \in \mathcal{P}$ we associate a local hardware clock H_{p_i}. It generally consists of an oscillator and a counting register that is incremented by the ticks of the oscillator. While clocks are discrete, all algorithms assume that clocks run continuously, i.e., H_{p_i} is a continuous function on some real-time interval. Moreover, although hardware clocks can drift apart from real time due to temperature changes and aging, it is commonly assumed that their drift is within a narrow envelope of real-time, as expressed below:

Assumption 1 (ρ-bounded clock) *For a very small known constant $\rho > 0$, we define a hardware clock $H_{p_i}(t)$ to be ρ-bounded provided that for all real time t:*

$$\frac{1}{(1+\rho)} \leq \frac{dH_{p_i}(t)}{dt} \leq (1+\rho)$$

with $H_{p_i}(t)$ denoting the value of the hardware clock of processor p_i at real-time t. Note that throughout this paper, lower case letters represent real times and upper case letters clock times.

Clock synchronization algorithms do not directly synchronize local hardware clocks. Rather, *logical clocks* are introduced. The value of a logical clock at real-time t, denoted $L_{p_i}(t)$, is determined by adding an adjustment term, denoted $A_{p_i}(t)$, to the local hardware clock $H_{p_i}(t)$. The adjustment term $A_{p_i}(t)$ can be either a discrete value, changed at each resynchronization [SC90, ST87] or a linear function of time [SC90, Cri89]. The *discrete clock adjustment* technique may cause a logical clock to instantaneously leap forward or be set back, and then continue to run at the speed of its underlying hardware clock. Many distributed applications requiring clock synchronization are not able to tolerate such behavior. This problem is avoided by well dimensioning the adjustment term $A_{p_i}(t)$ [ST87], or by using a linear function of time for it.

Figure 1 illustrates the concepts of hardware and logical clocks. It depicts three hardware clocks (a *fast* clock, a *slow* clock and a *perfect* clock) and two logical clocks, updated respectively using *discrete* and *linear* adjustment.

In order to be synchronized with respect to real time, i.e., to solve the *external* synchronization problem, some processors use the GPS (Global Positioning System) or radio receivers to obtain the time signal broadcast by a standard source of time, as the UTC (Universal Time Coordinated). Such processors are called *reference clocks*. In external clock synchronization algorithms, every reference time processor p_r has a hardware clock H_{p_r} approximating real time at any point within some a priori given error Δ (see assumption 2 below).

Assumption 2 (Bounded external deviation) *A reference clock H_{p_r} is correct at time t if $\mid H_{p_r}(t) - t \mid \leq \Delta$*

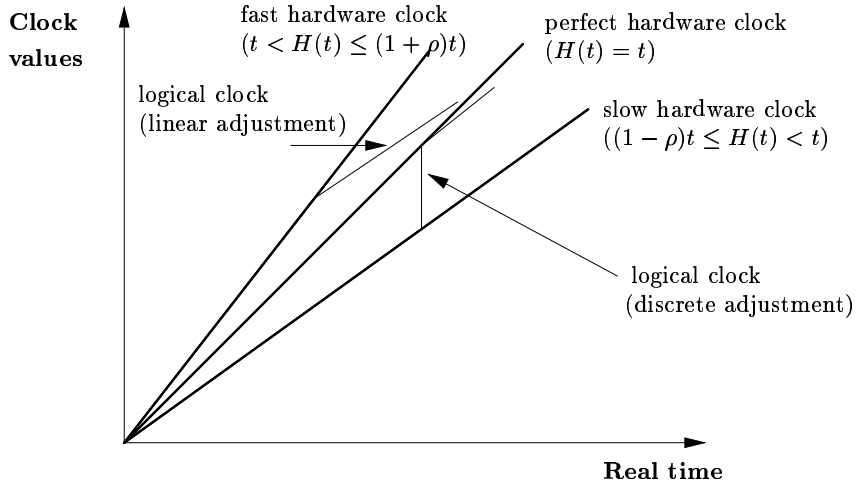

Figure 1: Hardware clocks and logical clocks

2.2 Network synchrony

In distributed real-time systems, message delays may be more or less predictable depending on the type of network used and the assumptions made on the network load. Some designers assume that known lower and upper bounds to deliver (i.e. to send, transport and receive) a message exist. This assumption is formalized below:

Assumption 3 (Bounded transmission delay) *The real time delay to send, transport, and receive any message over any link is within some known bounds* $[\delta - \epsilon, \delta + \epsilon]$*, with* $\delta - \epsilon \geq 0$.

When such an assumption holds, clock synchronization algorithms ensure that all correct logical clocks are within a maximum distance from each other (this distance is called the precision of the algorithm). Algorithms guaranteeing a precision are called *deterministic* clock synchronization algorithms.

On the other hand, it can be assumed that such tight bounds do not exist, for instance due to traffic overload. According to designers, it may be assumed either that message delays are modeled as a variable with arbitrary values, or that the mean and standard deviation of transmission delays are known. In the former case, we are dealing with *probabilistic* clock synchronization algorithms, although the assumptions made in the latter case are those of *statistical* clock synchronization algorithms. In both cases, the precision of any two correct clocks can only be guaranteed with some non-null probability.

2.3 Failure mode

A failure can be defined as a deviation from a *correct* behavior. All components (processors, communication links, and clocks) may commit failures. Following is a list of types of processor, link and clock failures that have mostly been assumed throughout clock synchronization algorithms.

Concerning clocks, most of the algorithms [HSSD84, ST87, LL88, LMS85, VCR97, PB95, MS85, FC95a, FC97] assume uncontrolled failures (also called Byzantine or arbitrary failures) [LSP82]. Other ones, such as [GZ89] algorithm assume timing failures [CASD85], a more restricted failure mode prohibiting conflicting information.

Clock timing failure: *A local hardware clock commits a timing failure if it does not meet assumption 1, i.e., is not ρ-bounded.*

Clock Byzantine failure: *A local hardware clock commits a Byzantine failure when it gives inaccurate, untimely or conflicting information. This includes dual-faced clocks, which may give different values of time to different processors at the same real-time.*

The failure semantics of processors assumed in published algorithms cover nearly all the kinds of failures ever identified. More precisely, processors may crash as in [VCR97, CAS86, GZ89] and [OS94], commit performance failures as in [VCR97, CAS86] and [Cri89] or more generally, commit Byzantine failures as in [HSSD84, ST87, LL88, LMS85, PB95, MS85, FC95a, FC97, MO83].

Processor crash failure: *A processor commits a crash failure if it behaves correctly and then stops executing forever (permanent failure).*

Processor performance failure: *A processor commits a performance failure if it completes a step in more than the specified value.*

Processor Byzantine failure: *A processor commits a* Byzantine *failure if it executes uncontrolled computation.*

With regard to the network, whatever its type (broadcast or point-to-point), the failure semantics are more restricted. A link may commit omission or performance failures as assumed in [CAS86, GZ89, VCR97, HSSD84], but never be partitioned.

Link omission failure: *A link l from a processor p_i to a processor p_j commits an omission failure on a message m if m is inserted into p_i's outgoing buffer but l does not transport m into p_j's incoming buffer.*

Link performance failure: *A link commits a performance failure if it transports some message in more time than its specified bound. Clearly, this applies only to systems with known upper and lower bounds on network latencies.*

Recall that the message delay consists of the time needed for the sender to send the message, to transport it, and for the receiver to deliver it. Thus a violation of assumption 3 may be due to a performance failure of the sender, the receiver, or the link between them.

If a component (clock, processor or link) commits a failure, we say that it is *faulty*; otherwise it is *correct*. Most algorithms assume a bound on the total number of faulty components (processors, clocks, network), as stated in assumption 4.

Assumption 4 (Bounded number of faulty components) *There are at most f faulty components in the system during the clock synchronization protocol.*

3 The clock synchronization problem

Given the system model presented in section 2, a clock synchronization algorithm aims at satisfying the following property for all correct clocks i and j and all real time t:

Property 1 (Agreement) $\mid L_{p_i}(t) - L_{p_j}(t) \mid \leq \gamma$, *where γ is called the* precision *of the clock synchronization algorithm.*

This property informally states that correct logical clocks are approximately equal. All deterministic clock synchronization algorithms meet this property, contrary to probabilistic and statistical ones that satisfy this property with a probability strictly less than one.

While the agreement property is obviously required, it is not a sufficient condition. For example, this property holds even if all logical clocks are simply set to zero and stopped. In order to rule out such trivial solutions to the clock synchronization problem, [LMS85] introduced the property given below:

Property 2 (Bounded correction) *There is a small bound Σ on the amount by which a correct clock is changed at each resynchronization.*

Another way to rule out trivial solutions is to require that logical clocks are permanently within a narrow envelope of real time. This property is called *accuracy* in most algorithms, and is called *bounded drift rate* in [FC97].

Property 3 (Accuracy) *For any correct processor $p_i \in \mathcal{P}$ and for real time t there exists a constant ν, called accuracy, such that for any execution of the clock synchronization algorithm $(1+\nu)^{-1}t + a \leq L_{p_i}(t) \leq (1+\nu)t + b$, with a and b some constants depending on the initial condition of the algorithm.*

This property is equivalent to the $(\alpha_1, \alpha_2, \alpha_3)$-validity property introduced by Lundelius and Lynch in [LL88]. The best drift rate of logical clocks achievable (and achieved in [ST87]) is equal to the underlying hardware clock drift ρ (*optimal accuracy*).

Finally, deterministic external clock synchronization algorithms have to bound the deviation between a correct logical clock and real time by an a priori given constant φ. This is expressed in the following property:

Property 4 (Bounded external deviation) *For any correct processor $p_i \in \mathcal{P}$, and any real time t, $\mid L_{p_i}(t) - t \mid \leq \varphi$.*

Note that in order to synchronize clocks, some algorithms may add the following assumption concerning clocks initial synchronization (see [LL88]):

Assumption 5 (Initial synchronization) *For any two correct processors p_i and p_j, if T_0 is the logical time at which p_i and p_j start the synchronization algorithm, then $|l_i(T_0) - l_j(T_0)| \leq \beta$, with $l_i(T_0)$ (resp. $l_j(T_0)$) the real-time when p_i (resp. p_j) logical clock shows T_0, and β a real-time value.*

4 A taxonomy of clock synchronization algorithms

This section is devoted to the proposed classification of clock synchronization algorithms. We classify clock synchronization algorithms according to two orthogonal features: their internal structure (section 4.1) and the basic building blocks from which the algorithms are built (section 4.2). Three building blocks have been identified and correspond to the three successive steps executed by every clock synchronization algorithm. Building blocks are generic in the sense that they apply to all clock synchronization algorithms — deterministic, probabilistic and statistical; internal and external — and are designed to support faulty components. We illustrate our taxonomy on a large set of clock synchronization algorithms in section 4.3.

4.1 Structure of the clock synchronization algorithm

Clock synchronization algorithms may be either symmetric or asymmetric. In symmetric algorithms, all processors play the same role, while in *asymmetric* algorithms one predefined processor, called the *master* has a specific role. The symmetry of the algorithm mainly influences its ability to support failures, and its cost in term of number of messages exchanged to achieve the sought precision. In some clock synchronization algorithms [MS85, FC95a, FC97, MO83], and [LMS85] (algorithms CON and COM), the choice of the structure of the algorithm is left to its designer.

4.1.1 Symmetric schemes

In *symmetric* algorithms, each processor plays the same role and executes the whole clock synchronization algorithm. Each processor $p_i \in \mathcal{P}$ disseminates messages (containing for

instance the current value of its logical clock) to every other one, and uses the messages received from the remote processors to correct its local clock. Symmetric algorithms can be split in two classes, *flooding-based* and *ring-based*, depending on the virtual path taken for transmitting a message from one processor to every other one.

Flooding-based scheme

In *flooding-based* symmetric algorithms, each processor sends its messages to all outgoing links. Messages received on incoming links are relayed when a non fully connected network is used. A flooding-based technique is used in the algorithms [CAS86, HSSD84, ST87, LL88, VCR97, PB95] and [LMS85] (algorithm CSM). The benefit of flooding-based techniques is their natural support for fault tolerance, i.e., they do not exhibit a single point of failure. However, they may require up to n^2 messages to disseminate a message to all processors in the system, with n the number of processors in the system. This large number of messages can be lowered to n if a broadcast network is used [VCR97].

Virtual ring scheme

The virtual ring scheme was designed in order to decrease the number of exchanged messages experienced in the flooding-based schemes. In the ring scheme, all the processors in the system are gathered along a virtual ring. The number of messages is reduced by sending only one message along this cyclic path. As this message travels on the ring, each processor adds its own data to the message [OS94]. Compared with flooding-based schemes, virtual ring schemes need a smaller number of message exchanges (only n messages per resynchronization are used), but need extensions in order to support processor failures.

4.1.2 Asymmetric schemes

In asymmetric schemes, also called *master-slave* schemes, processors involved in clock synchronization play a different role. One processor in the system is designed as the *master*, and the other processors are designed as the *slaves*. Within this scheme, two techniques exist. In the first one, called the *master-controlled* scheme, the master acts as a coordinator of the clock synchronization algorithm. It collects the slaves clocks, estimates them and sends them back the corrected clock values. Such a technique has been used by Gusella et al. in [GZ86]. In the other technique, commonly called *slave-controlled* scheme, the master acts as a reference processor providing time from which all the slaves resynchronize [Cri89, Arv94].

The obvious advantage of asymmetric schemes is their low cost in term of number of messages exchanged. On the other hand, the presence of the master represents a single point of failure, and needs some extra mechanisms to be avoided, such as fault detection

followed by the election of a new master, or duplication of masters. Another drawback of the asymmetric scheme is the presence of a single master that can be swamped by large numbers of synchronization messages, thus invalidating in some way communication delays assumptions.

4.2 Clock synchronization building blocks

In any clock synchronization algorithm — deterministic or not, external or not, and whatever its structure is — each processor $p_i \in \mathcal{P}$ has to achieve three successive steps. First, it has to detect the instant at which it must resynchronize, which is done by invoking the *resynchronization event detection block* (section 4.2.1). Second, it has to estimate the value of remote logical clocks, which is achieved by invoking the *remote clock estimation block* (section 4.2.2). Third, it has to apply a correction on its local clock according to the result of the second step. The *clock correction block* (section 4.2.1) is devoted to this task. Investigation of existing clock synchronization algorithms led us to the identification of a palette of techniques implementing these building blocks. Unless explicitly stated, all the techniques implementing each building block are suited to all types of algorithms (deterministic, probabilistic or statistical, internal or external, symmetric or not).

4.2.1 Synchronization event detection block

The objective of p_i's *synchronization event detection block* is to trigger a *resynchronization event* informing processor p_i to start the clock synchronization algorithm. Indeed, due to clock drift, clocks must be re-synchronized often enough to guarantee *Agreement* (property 1). A common way to resynchronize clocks is to periodically initialize the clock synchronization algorithm, and thus to consider the algorithm as a *round-based* algorithm, each round being devoted to the resynchronization of all clocks. The difficulty arises when dealing with the time at which rounds must start. So far, two techniques have been devised. The first one used in [LL88, LMS85, PB95, TFKW89] relies on initially synchronized clocks. The second one adopted in [CAS86, HSSD84, ST87, VCR97] uses message exchanges. These two techniques are presented in the following two paragraphs. Remark that a large number of algorithms [MS85, FC95a, FC97, MO83] do not detail how rounds are detected.

Detection technique based on initially synchronized clocks

The mostly used technique is to use initially approximately β-synchronized clocks (assumption 5) to detect round boundaries. When using such a technique, processor p_i considers that a new round k begins when its logical clock reaches time kR, where R is the round duration, i.e., the time between two successive resynchronizations measured in logical time units.

Intuitively, to keep logical clocks as closely synchronized as possible, β and R must be as small as possible. However, Lundelius and Lynch prove in [LL88] that R cannot be arbitrarily small to be able to distinguish between successive rounds. For instance, in [LL88], R must satisfy the following inequation: $2(1+\rho)(\beta+\epsilon)+(1+\rho)\max(\delta,\beta+\epsilon)+\rho\delta < R \leq \beta/(4\rho)-\epsilon/\rho-\rho(\beta+\delta+\epsilon)-2\beta-\delta-2\epsilon$.

Detection technique based on message exchanges

As mentioned in [LMS85], the precision of algorithms using approximately synchronized clocks to detect round boundaries is very sensitive to the initial clock synchronization value β. Another way to detect the beginning of a round consists for a processor to send a message to all processors in $\mathcal{P}$ as soon as its logical clock reaches the predefined value kR. Upon receipt of a fixed number of such messages, a processor $p_i \in \mathcal{P}$ starts a new clock synchronization round. The number of expected messages depends on the maximum number of failures assumed ($f+1$ or $2f+1$ depending on whether digital signatures are used or not [ST87]).

Clearly, as rounds are triggered upon message receipt the precision of algorithms depends on message latencies [CAS86, HSSD84, ST87]. By using broadcast networks, exhibiting a small variance of transmission delays, precision can be improved [VR92].

4.2.2 Remote clock estimation block

The objective of the *remote clock estimation* block is to get some knowledge about the value of remote clocks. Indeed, due to the presence of non-null and variable communication delays and due to the existence of clock drifts, getting an exact knowledge of a remote clock value is not feasible. Thus, only *estimates* of remote clock values can be acquired. Processor p_i's *remote clock estimation* block is triggered upon receipt of the resynchronization event generated by p_i's *resynchronization event detection* block. The output of the *remote clock estimation* block is a *clock estimation message* containing the set of remote clock estimates. This message will trigger p_i's *clock correction* block.

Two techniques can be selected to estimate remote clock values. These techniques, named *Time Transmission* (TT) and *Remote Clock Reading* (RCR) are described hereafter. Some algorithms ([FC95a, FC97, MS85] and [LMS85] – algorithms CON and COM) deliberately do not restrict their algorithm to work with one of these techniques. They only assume every remote clock can be estimated with a small bounded error Θ, the error obviously depending on the actual clock estimation technique, TT or RCR, used.

The time transmission (TT) technique

In the *time transmission* (TT) technique, processor $p_i \in \mathcal{P}$ sends its local clock in a message. The receiving processor $p_j \in \mathcal{P}$ uses the information carried in this message to estimate p_i's clock. Depending on whether or not communication delays are bounded (assumption 3), two variants exist.

The first variant is suited to systems with bounded communication delays, and requires clocks to be initially β-synchronized (assumption 5). In this variant, described in [LL88, DHSS95] and [CAS86], processor p_j sends its message to all processes in $\mathcal{P}$ at a fixed local logical time, say T. Meanwhile, processor p_j collects messages from the other processors[1] within a particular amount of time measured on its logical clock. Processor p_j stores the local time at which these messages are received and builds its *clock estimation messages* set and sends it to its local *clock correction* block. The interval of time during which p_j waits for the message is chosen just large enough to ensure that p_j receives it in case where p_i is non faulty. The length of this interval equals $(1+\rho)(\delta+\epsilon+\beta)$. At receipt time, p_i's clock belongs to the interval $[T+(1-\rho)(\delta-\epsilon-\beta), T+(1+\rho)(\delta+\epsilon+\beta)]$. The lower bound (resp. upper bound) of this interval is obtained when (i) the sending processor p_i reaches time T β time units before (resp. after) p_j does; (ii) p_i's clock drift rate equals $1-\rho$ (resp. $1+\rho$); (iii) the communication delay between p_i and p_j, measured in real-time units, equals $\delta-\epsilon$ (resp. $\delta+\epsilon$). Any value belonging to this interval can be used as an estimate of p_i's clock. For instance, in [LL88] the message receipt time according to the logical clock of p_j is used to estimate p_i's clock. The maximum estimation error one can make when using TT to estimate a remote clock equals $2(\epsilon+\beta+\rho\delta)$ (the smaller estimation error is given by the midpoint of the above interval).

In case where communication delays are not bounded, a statistical variant to TT has been identified [OS94, Arv94]. In this variant, the absence of both initial synchronized clocks and precise upper bounds on the transmission delays are overcome by s successive transmissions of timestamped synchronization messages, with $s \geq 1$. More precisely, this variant consists for processor p_i to send a sequence of s messages timestamped with its clock value to processor p_j. Then p_j estimates p_i's clock time on the basis of message timestamps and message delays statistics. If T_l is the timestamp included by p_i in its l^{th} message, R_l the receipt time of the l^{th} at p_j, and $\overline{\delta}$ be the expected value of the message delay, then the estimated value of p_j's clock at the receipt of the s^{th} message by p_i is $E_s = R_s + \overline{\delta} - \frac{1}{s}\sum_{l=1}^{s}(R_l - T_l)$. Note that this technique requires the knowledge of the expectation and deviation of transmission delays.

Lots of deterministic clock synchronization algorithms [CAS86, HSSD84, ST87, LL88, PB95] and [LMS85] (algorithm CSM) and statistical ones [OS94, Arv94] rely on the TT

[1] according to the structure of the algorithm (symmetric or asymmetric, master-controlled or slave-controlled) and to the failure mode assumed, the number of messages p_j waits for differs.

technique because of its low cost in terms of messages compared with the *RCR* technique described hereafter.

Remote clock reading (RCR) technique

The *remote clock reading* technique (RCR) has been introduced by [Cri89] to circumvent the absence of knowledge of an upper bound of communication delays. The *RCR* technique works as follows. A processor p_j willing to estimate the clock of a remote processor p_i, sends a request message to p_i, and waits for a certain amount of time p_i's response. The response contains the current value of p_i's logical clock, say T. Let D be half of the round trip delay measured on p_j's clock between the sending of p_j's message and receipt of p_i's response. The interval in which p_i's clock belongs to when p_j receives the message is $[T + (\delta - \epsilon)(1 - \rho), T + 2D(1 + 2\rho) - (\delta - \epsilon)(1 + \rho)]$. This interval is obtained by taking into account (i) the distribution of communication delays, (ii) the drift of p_i's clock during the estimation process, (iii) the drift of p_j's clock during the estimation process (see [Cri89] for more details). Any value belonging to this interval can be used to estimate p_i's clock. For instance in [GZ89], value $T + D$ is used to estimate p_i's clock. The maximum clock reading error when using *RCR* to estimate a remote clock is equal to $2D(1 + 2\rho) - 2(\delta - \epsilon)$. The clock reading error is divided by a factor 2 if the mean value of the estimation interval is selected as remote clock estimate. The choice of the estimate value, as well as the building and the sending of the *clock estimation messages* set are achieved as in the *TT* technique.

Note that the smaller D is, the smaller the length of the estimation interval is, and thus the better the remote clock estimate is. Probabilistic clock synchronization algorithms use this property to obtain a low and known reading error. If p_i wants to have a clock reading error, say Θ, better than the one obtained with a $2D$ round trip delay, it sends again its message as long as the response from p_j arrives in a minimum of time, say $2U$ ($U < D$), with $U = (1 - 2\rho)(\Theta + \delta - \epsilon)$. This value U must be as close as possible to the minimum transmission delay $\delta - \epsilon$ to get a good reading error but greater than $(\delta - \epsilon)(1 + \rho)$ to ensure that between the sending of a message and its reception there is a real time delay of at least $\delta - \epsilon$.

A variant of the *RCR* technique is used in the deterministic algorithm described in [MO83]. In this variant, p_i's clock is stored by a tuple (L_{p_i}, e_{p_i}), where L_{p_i} is p_i's logical clock and e_{p_i} is the maximum error assumed by p_i on L_{p_i}. In this variant, the response provided by processor p_i is $(L_{p_i}(T), e_{p_i}(T))$. This leads to the maximum clock error reading equal to $e_{p_i}(T) + (1 + \rho)D$.

The *RCR* clock estimation technique, used in [Cri89, GZ89, OS94, MO83, VCR97]. Its benefits are twofold. First its correctness does not depend on any assumption about the particular shape of the message delay distribution function. Second its correctness does not rely on approximately synchronized clocks. Furthermore, by using this technique, a

processor exactly knows the error it makes when estimating a remote clock, and thus knows when it has lost synchronization with the other processors. However, twice more messages than the *TT* technique are required due to the use of a request-reply approach.

4.2.3 Clock correction block

The final step in a clock synchronization algorithm is to adjust a clock in relation with the other ones. This is done by the *clock correction* block. The input of p_i's *clock correction* block is *pi*'s clock *estimation messages* set. This block has no output as its effect is the correction of p_i's logical clock. To correct a clock , one can use the clock estimates contained in the input message or can simply use the presence of such a message to compute the corrected clock value. In the former case, the corrected clock value is obtained by computing some kind of averaging of remote clock estimates (*convergence-averaging* techniques), while in the latter case (*convergence-nonaveraging* techniques) the corrected value only depends on the receipt of a fixed number of remote clock values.

Convergence-averaging techniques

Convergence-averaging techniques, also called *convergence function based* techniques, use a set of remote clock estimates to compute a corrected clock value. These techniques use a so-called *convergence function*, which takes as arguments a set of clock estimates and returns a single clock value. An exhaustive list of convergence functions introduced before 1987, is given in [Sch86]. Convergence functions compute some kind of averaging on clock estimates; some of them are designed to tolerate erroneous clock estimates. In the following, the notation $f(p_i, x_1, \cdots, x_n)$ identifies a convergence function. Argument p_i is the processor requesting clock correction, and variables $x_1, \cdots, x_n$ are estimated clock values. The result of the convergence function $f(p_i, x_1, \cdots, x_n)$ is the corrected clock value for processor p_i.

There are many possible convergence functions, whose mostly referenced are listed below. For the following six convergence functions (f_m, f_e, f_{fc}, f_{ftm}, f_{dftm}, and f_{sw}) every x_i is an integer value. For the last two ones (f_{mm} and f_{im}) every x_i is an interval.

- *Interactive convergence function.* This function, also known as the *Egocentric average* function f_e is used in [LMS85]. Convergence function $f_e(p, x_1, \cdots, x_n)$ returns the average of all arguments x_1 through x_n, where x_j $(1 \leq j \leq n)$ is kept intact if it is no more than ϖ from x_p and is replaced by x_p otherwise. The constant ϖ has to be selected appropriately. The condition $\varpi > \gamma$, where γ is the achievable precision of clocks, must hold to avoid *clustering* of the clocks, i.e., slow clocks synchronized only to the other slow clocks while all the fast clocks synchronize only to the other fast clocks. On the other hand, a large ϖ leads to a reduced quality of the precision as faulty clocks values are not rejected. The advantage of algorithms using f_e is that

no sorting algorithm must be applied to reject faulty clock values, thus reducing the complexity of clock correction.

- *Fast convergence function.* This function used in [MS85, GZ86], and denoted f_{fc}, returns the average of all arguments x_1 to x_n that are within ϖ of at least $n-f$ other arguments. Although f_{fc} yields high quality precision [MS85, GZ86], the complexity of computing all time differences to all the other processors of the system is quite important.

- *Fault tolerant midpoint function.* This function, used in [LL88] and denoted f_{ftm}, returns the midpoint of the range of values spanned by arguments x_1 to x_n after the f highest and the f lowest values have been discarded. Convergence function f_{ftm} is based on the assumption that faulty clocks run too fast or too slowly, and thus that good clocks generally lie between.

- *Differential fault-tolerant midpoint function.* This function denoted f_{dftm}, has been introduced in [FC95b] and used in [FC95b, FC97]. Convergence function f_{dftm} was proven to be optimal with respect to the best precision achievable and the best accuracy (ρ) achievable for logical clocks (which is not the case with the f_{ftm} function). This function is defined as follows:

$$f_{dftm}(p, x_1, \cdots, x_n) = \frac{\min(T-\Theta, x_l) + \max(T+\Theta, x_u)}{2},$$
$$\text{where } x_l = x_{h_{f+1}}, x_u = x_{h_{n-f}}$$
$$\text{with } x_{h_1} \leq x_{h_2} \leq x_{h_n}, h_p \neq h_q;\ 1 \leq\ h_p, h_q \leq n$$

  where T is p's logical time and Θ is the maximum reading error of a distant clock.

- *Sliding window function.* This function, proposed in [PB95], selects a fixed size window w that contains the larger number of clock estimates. Two convergence functions are proposed; they differ by the way a window is chosen when multiple windows contain the same number of clock estimates, and by the way the correction term is computed once the window has been identified. The first function, called f^{det}_{mean}, chooses the first window, and returns the mean of the clock values contained in the window instance. The second function, called f_{median}, chooses the window containing clock estimates having the smallest variance, and returns the median of all clock estimates within the selected window. The main interest of sliding window convergence functions is that logical clocks closeness degrades gracefully when more failures than assumed occur.

- *Minimization of the maximum error interval-based function.* This function takes for each x_i an interval $[L_{p_i}(t) - e_{p_i}(t), L_{p_i}(t) + e_{p_i}(t)]$, where $e_{p_i}(t)$ is the maximum error

on p_i's clock estimate, and returns an interval for the corrected clock value. More precisely:

$$S_p = \{x_i \mid \text{consistent}(p, p_i)\}$$
$$f_{mm}(p, x_1, \cdots, x_n) = [L_p(t) - e_p(t), L_p(t) + e_p(t)], where\, \forall i \in S_p, e_p(t) \leq e_{p_i}(t)$$

where *consistent*(p, p_j) is true if $\mid L_p(t) - L_{p_j}(t) \mid \leq e_p(t) - e_{p_j}(t)$.

- *Intersection of the maximum error interval-based function.* This function, denoted f_{im}, is similar to f_{mm} in the sense they both take intervals representing clock estimates as arguments. However, function f_{im} returns an intersection of the intervals of the clock estimates. Both functions are used in [MO83].

Notice that the input of the clock correction block can be reduced to a single clock estimate. In that case, the result of the convergence function (f_{id}) is simply a copy of this clock estimate. This can be used for every processor in asymmetric schemes to be synchronized with respect to a single time reference, as in [Cri89] and [Arv94].

Convergence-nonaveraging techniques

Contrary to convergence-averaging techniques, which compute a new clock value by using the contents of clock estimation message, convergence-nonaveraging techniques only use the fact that a fixed number of estimates of remote clocks have been received to compute a new clock value. The number of expected clock estimates depends on the type and the number of tolerated failures. For instance, the clock correction block has to wait for $2f+1$ estimates to support f Byzantine failures in a system not using digital signatures, while in case using signed messages, only $f+1$ estimates are required. In [CAS86], only one estimate is required as only performance failures are supported.

When all the required clock estimates are received, the clock correction block updates the logical clock with a value that differs according to algorithms. In [ST87], the logical clock is corrected with value $kR + \pi$, where k is the round number, R is the round duration and π is a constant. As mentioned in section 2, π should be large enough to avoid clocks to be set backwards. In [CAS86, HSSD84], the logical clock is simply corrected with kR. In [VCR97], it is corrected with the value of one of the clock estimates contained in the clock estimation message.

4.3 Illustration of the taxonomy

This section illustrates our taxonomy on a long (but not exhaustive) list of the most referenced deterministic, probabilistic and statistical clock synchronization algorithms. We present in section 4.3.1 the structure and building blocks of all algorithm. In section 4.3.2, the properties and cost of every algorithm is given.

4.3.1 Classification of clock synchronization algorithms

This section presents the classification of the most referenced clock synchronization algorithms. Table 1 gives for each algorithm its type, according to the synchrony model defined in section 2.2; the failure mode assumed for each component, as defined in section 2.3; its structure as introduced in section 4.1; its building blocks as defined in section 4.2.

Reference	**Type** [a]	**Failures** [b]			**Structure** [c]	**Synchronization Event Detection** [e]	**Remote Clock Estimation**	**Clock Correction** [d]
		C	L	P				
[CAS86]	D	R	P	O/P	SYM - FLOOD	MSG	TT	NAV
[HSSD84]	D	B	P	B	SYM - FLOOD	MSG	TT	NAV
[ST87]	D	B	R	B	SYM - FLOOD	MSG	Empty	NAV
[LL88]	D	B	R	B	SYM - FLOOD	SYNC	TT	AV - f_{ftm}
[LMS85] (CSM)	D	B	R	B	SYM - FLOOD	SYNC	TT	AV - f_e
[VCR97]	D	B	O	C/P	SYM - FLOOD	MSG	RCR	AV - f_m
[PB95]	D	B	R	B	SYM - FLOOD	SYNC	TT	AV - f^{det}_{mean}
[GZ89] (Tempo)	D	T	P	C	ASYM - MAST	not required	RCR	AV - f_{fc}
[MS85]	D	B	R	B	not imposed	not imposed	not imposed	AV - f_{fc}
[FC95a]	D	B	R	B	not imposed	not imposed	not imposed	AV - f_{dftm}
[FC97]	D	B	R	B	not imposed	not imposed	not imposed	AV - f_{dftm}
[LMS85] (CON)	D	B	R	B	not imposed	not imposed	not imposed	AV - f_e
[LMS85] (COM)	D	B	R	B	not imposed	not imposed	not imposed	AV - f_e
[MO83] (MM)	D	R	R	B	not imposed	not imposed	RCR	AV - f_{mm}
[MO83] (IM)	D	R	R	B	not imposed	not imposed	RCR	AV - f_{im}
[OS94] (TT)	S	R	—	C	SYM - RING	not imposed	TT	not considered
[OS94] (RCR)	P	R	—	C	SYM - RING	not imposed	RCR	not considered
[Arv94]	S	R	—	R	ASYM - SLAV	not imposed	TT	AV - f_{id}
[Cri89]	P	T	—	P	ASYM - SLAV	not imposed	RCR	AV - f_{id}

Table 1: Classification of clock synchronization algorithms

[a] D stands for deterministic, P for probabilistic and S for statistical.

[b] Components are labeled C (Clock), L (Link) and P(Processor). R stands for Reliable, C for Crash, P for Performance, B for Byzantine, and T for Timing. For probabilistic and statistical algorithms, no upper bound on message delays is assumed. Consequently, omission and performance failures are irrelevant, which is indicated by sign —.

[c] Symmetric and asymmetric schemes are denoted by SYM and ASYM. FLOOD, RING, MAST and SLAV name flooding-based, ring-based, master-controlled and slave-controlled schemes.

[d] NAV is used for non averaging techniques. Averaging techniques are denoted by AV and the convergence function name is given.

[e] SYNC (resp. MSG) is used when rounds are detected thanks to initially synchronized clocks (resp. message exchanges)

Table 1 shows that all the clock synchronization algorithms, whatever their type, can be completely described with the three building blocks we have identified in section 4.2, and the techniques implementing them. This shows the completeness of our taxonomy. In addition, this table demonstrates that each technique implementing a given building block is not tied to a specific type of algorithm (for instance, the RCR technique is both used in

the deterministic asymmetric algorithm presented in [GZ89] and the statistical symmetric algorithm described in [Arv94]). This shows the modularity of our taxonomy.

4.3.2 Properties of clock synchronization algorithms

In this section, clock synchronization algorithms are presented under performance and cost features. More precisely, for each of them is given the class of synchronization problem it solves (internal clock synchronization (I), external (E) or both (I+E)), the redundancy degree needed to tolerate f faulty components, its precision, its accuracy and its cost in terms of message number to achieve the sought precision.

The results are given in two tables. Table 2 is devoted to performance of deterministic algorithms, while table 3 concerns probabilistic and statistical ones.

Note that the drift of hardware clocks (ρ) being very small, we make in table 2 and table 3 the commonly adopted assumption that the terms ρ^m for $m \geq 2$ can be neglected.

Table 3 shows that our concern in using a uniform notation provides a thorough comparison of the performance of all algorithms.

Reference	Type	Redundancy	Precision	Accuracy	Msg.
[CAS86] [a]	I	none	$(\delta+\epsilon)(1+\rho)$ $+2\rho(R(1+\rho)+(\delta+\epsilon))$	ρ	n^2
[HSSD84]	I	none	$(1+\rho)(\delta+\epsilon)+2\rho(1+\rho)R$	$(f+1)2\rho$	n^2
[ST87]	I	$2f+1$	$(\delta+\epsilon)[(1+\rho)^3+2\rho]+(1+\rho)R$	ρ	n^2
[LL88] [b]	I	$3f+1$	$5\epsilon+4\rho\delta+4\rho R$	$\rho+\frac{\epsilon}{R_{min}}$	n^2
[LMS85] (CSM) [c]	I	$2f+1$	$(f+6)\epsilon+6\rho S+2\rho R$	$(2f+12)\epsilon$ $+10\rho S+2\rho R$	n^{f+1}
[VCR97] [d]	I+E	$(f_0+1)(f_p+1)$	$\Gamma_t+2\rho\Gamma_a+2\rho R$	$\frac{\rho R-(1-\rho)\Gamma_t}{R+(1-\rho)\Gamma_t}$	$3n$ bcasts
[PB95]	I	$4f+1$	$\frac{(n^2+n-f^2)(\delta+\epsilon)+2R\rho(n-f)(n-f+1)}{n^2-5fn+n+4f^2-2f}$	not given	n^2
[GZ89] (Tempo) [e]	I	$2f+1$	$4D(1+2\rho)-4(\delta-\epsilon)+2\rho R$	not given	$3n$
[MS85]	I	$3f+1$	$\frac{(n+f)\Theta+2f(\pi+\Theta)}{n}+2\rho R$	not given	—
[FC95a]	I	$3f+1$	$4\Theta+4\rho R+2\rho\beta$	ρ	—
[FC97] [f]	I+E	$2f+1$	$\varphi=\Delta+\Theta+\rho R$	ρ	—
[LMS85] (CON)	I	$3f+1$	$max((\frac{n}{n-3f}\Theta+2\rho(R+2S\frac{n-f}{n})),$ $\beta+2\rho R)$	not given	—
[LMS85] (COM)	I	$3f+1$	$(6f+4)\Theta+2\rho S(4f+3)+2\rho R$	$(12f+8)\Theta+$ $(8f+5)2\rho S+2\rho R$	—
[MO83] (MM) [g]	I	$3f+1$	$2E_m(t)+2(\delta+\epsilon)$ $+2\rho(R+2(\delta+\epsilon))$	not given	—
[MO83] (IM)	I	$3f+1$	$\delta+\epsilon+2\rho R$	not given	—

Table 2: Properties of deterministic clock synchronization algorithms

[a] Although not required in [CAS86], we assume a fully connected network in order to be able to compare the algorithm precision with the one of other algorithms. The same remark applies to [HSSD84] and [ST87].

[b] The worst case precision given here is obtained when taking $\beta = 4\epsilon + 4\rho R$ as suggested by the authors. R_{min} is the minimum duration of a round (see section 4.2.1).

[c] S stands for the time interval during which clock estimates are obtained (last S seconds of R). The precision given here is the precision along the real time axis, and defines how closely in real-time clocks reach the same logical clock value. The same definition of precision applies to all the algorithms described in [LMS85], which make these algorithms hardly comparable with the others. The value given in the *Accuracy* column is the bound on clock correction at each resynchronization (requirement 2).

[d] In [VCR97], at most f_0 omissions are supported during each synchronization round, and there can be at most f_p faulty clock-node pairs per round. The algorithms takes advantage of broadcast networks, in which nodes receiving an uncorrupted message receive it at real time values that differ, at most, by a known small constant Γ_t. Γ_a is the maximum duration of an agreement protocol.

[e] The algorithm described in [GZ89] does not inherently support the failure of the master process. When the master process crashes, synchronization can be lost until a new master process is elected. The precision of the algorithm depends on the round-trip time $2D$, and equals at worst $8(\epsilon+\rho(\delta+\epsilon))+2\rho R$ when D equals $\delta+\epsilon$.

[f] It is assumed in [FC97] that reference time servers approximate real time with an a priori given error Δ (assumption 2), and that external time server do not drift.

[g] The basic requirement of the algorithm is that clocks are represented validity intervals. The algorithms are extended to support faulty clocks in [Mar83]. Variable $E_m(t)$ stands for the smallest clock error in the system. Note that the agreement property is ensured only if clocks are initially mutually consistent (see section 4.2.3).

Reference	**Type**	**Precision**	**Messages**
[OS94] (TT) [a]	I	$\int_{-\infty}^{\infty} dy \int_{-\infty}^{w+y+n(\delta-\epsilon)} f_{max}(y,m) f_{min}(x,m) dx$	not mentioned
[OS94] (RCR) [b]	I	$P_{\gamma<\gamma_{max}} = erf(\frac{\gamma_{max}\sqrt{2m}}{\sqrt{n}\mu})$	$m = 2\frac{n\mu^2}{2\gamma_{max}^2} erf^{-1}(P_{\gamma<\gamma_{max}})^2$
[Arv94] [c]	I	$\gamma_{max} = 2(\gamma_{synch} + (R+2\epsilon)\rho)$	$\frac{2\sigma_d^2 (erfc^{-1}(p))^2}{\epsilon_{max}^2}$
[Cri89] [d]	I	$U - \delta + \epsilon + \rho k(1+\rho)W$	$\frac{2}{1-p}$

Table 3: Properties of probabilistic and statistical clock synchronization algorithms

[a] In [OS94], f_{min} (resp. f_{max}) stands for the density function of the lower (resp. the upper) bound on the skew interval generated by a message, assuming that endpoints of an interval can be modeled as independent random variables. The given constant $w/2$ is the sought remote clock reading error. The value given in the *Precision* column is the probability a remote clock is estimated with an error $w/2$ using m messages.

[b] The value given in the *Precision* column is the probability a remote clock is estimated with an error lower than a given constant γ_{max} using m message exchanges. Function erf is the error function and is defined as $erf(u) = \frac{2}{\sqrt{\pi}} \int_0^u e^{-y^2} dy$. Variable γ is the difference between any two clocks assuming that the average of the transmission delay δ is known, γ_{max} the maximum distance the estimate is allowed to vary from the true value, and μ^2 the variance of a single hop on the ring. The value given in the column *Messages* is the average number of messages exchanged per resynchronization.

[c] The value given in the *Precision* column is the best precision achievable when an precision of γ_{synch} is obtained just after resynchronization. Variable p stands for the probability a precision of γ_{synch} is obtained, σ^2 is the standard deviation of the message delay, and $erfc^{-1}(p)$ is the inverse of the complementary error function defined as $erfc(u) = 1 - erf(u)$.

[d] The value given in the *Precision* column is the best precision achievable assuming that no failure occurs. Variable k is the number of successive reading attempts performed by the master, and W is the delay elapsed between each reading attempt. U is the maximal round trip delay accepted by the master to consider a slave response, beyond which, the master discards a reading attempt. The figure given in the *Messages* column is the average number of messages required in a fail-free environment; p is the probability that a process observes a round trip delay greater than $2U$.

5 Performance evaluation of clock synchronization algorithms

This section is devoted to an analysis of the behavior of a panel of deterministic clock synchronization algorithms [LL88, ST87, PB95, GZ89] constructed from the building blocks identified in our taxonomy. The algorithms behavior is analyzed through simulation, first in a failure-free environment and second in the presence of failures. Simulation results allow us to compare the algorithms *tightness* (i.e. the maximum difference between any two clocks) with their (worst-case) guaranteed precision, and allow us to compare the algorithms behavior in the presence of both expected and unexpected failures. Section 5.1 describes our simulation environment. Simulation results are given in Section 5.2.

5.1 Simulation environment

The simulation environment used for the performance evaluation is described in three steps. Section 5.1.1 first presents our simulation method in a failure-free environment. Section 5.1.2 then details how various types of failures – ranging from crash to Byzantine failures – are simulated. Finally, Section 5.1.3 details the characteristics of the fault-tolerant clock synchronization algorithms we have evaluated.

5.1.1 Simulation in a failure-free environment

We simulate a system of n processors connected through a point-to-point fully connected network. Each entity we wish to simulate (clock, network and the three building blocks of every clock synchronization algorithm) is assigned to an independent lightweight thread of control (*process* of the CSIM simulation library [Sch92]). A single instance of the network process is created, while n instances of the other processes exist.

The clock process maintains a double precision floating point value (current clock value). An increment is added to the current clock value at every clock tick (the duration of the clock tick is embodied by the simulation parameter *granularity*), the value of the increment depending on the clock drift rate (simulation parameter ρ_i).

The network process loops waiting for requests to exchange messages between processes. Every time a message transmission is requested, the network process waits for a delay belonging to the interval $[\delta - \epsilon, \delta + \epsilon]$ before delivering it to its recipient. We make message transmission delays follow a *normal* law within this interval.

Each of the three processes assigned to the simulation of the clock synchronization building blocks (resynchronization detection, clock estimation and clock correction) executes one of the detection/estimation/correction method identified in Section 4.2. We neglect the computation time of these building blocks.

Name	Description	Value
n	Number of processors	8
granularity	Clock granularity	1e-7 s
$\rho_1, ..., \rho_n$	Clocks drifts	8e-6,-8e-6,5e-6,-5e-6,1e-6,-9e-6,2e-6,-3e-6
ρ	Maximum clock drift ($\max(\rho_i)$)	1e-5
δ	Mean network latency	8 ms
ϵ	Variation of network latency	0.1 ms
R	Duration of a synchronization round	1 mn

Table 4: Simulation parameters in a failure-free environment

Table 4 gives the numerical values of the simulation parameters used in the experiments. These values are representative of a 10 Mb Ethernet local area network of workstations.

5.1.2 Simulation in a failure-prone environment

In our simulation model, clocks experience crash, timing and Byzantine failures depending on the simulated algorithm. A clock crash is simulated by stopping the incrementation of the current clock value, and since a crashed clock cannot be used by its processor to keep the other clocks synchronized, this processor stops executing the clock synchronization algorithm. That is, in our simulation model, a clock crash is equivalent to a processor crash. A clock timing failure is simulated by changing the clock drift rate to a value randomly selected in the intervals $[-4\rho, -2\rho]$ or $[2\rho, 4\rho]$. We simulate a clock Byzantine failure by setting the current clock value to a value randomly chosen within the interval $[-100000, 100000]$, this interval being large enough to simulate any kind of clock behavior (see section 2.3).

Communication links may suffer from omission and performance failures depending on the algorithm. An omission is simulated simply by making the network process not deliver the message to its recipient. A performance failure is simulated by making it deliver a message after a delay randomly selected within the interval $[\delta + \epsilon, 4\delta]$.

Processors may crash. Once a processor has crashed, its clock synchronization building blocks stop interacting with the building blocks located on other processors (i.e., stop sending and processing clock synchronization messages). Instead of simulating processor Byzantine failures, we consider they are equivalent to clock Byzantine failures.

For every class of failure (i.e., clock crash, timing and Byzantine failures, network omission and performance failures, processor crashes) we specify the number *nf* of failures of this class that will occur in the whole system during an interval I. Clock and processor crashes, clock timing failures and clock Byzantine failures are assumed to be *permanent* failures, and for these types of failures the length of I is equal to the simulation duration (we do not deal with reintegration of components affected by a permanent failure). Link omissions and link performance failures are assumed to be *transient* failures. Thus, for link failures the

length of I is equal to the duration of a round. If nf is the number of failures of type t per interval I, I is divided into nf sub-intervals and exactly one failure of type t occurs in every sub-interval. The faulty component is selected at random.

Finally, we specify for every algorithm the number f of failures the algorithm is able to support according to the failure assumption it makes (in the case the failure assumptions of a given algorithm are met, f is equal to the sum of nf for every class of failure).

5.1.3 Selected algorithms

Our performance evaluation is based on the simulation of five deterministic clock synchronization algorithms, namely the algorithms proposed in [GZ89, LL88, PB95, ST87] and MsgRcrFTM algorithm[2]. These algorithms were chosen because they cover a large spectrum of associations of techniques implementing the building blocks identified in Section 4.2. The characteristics of these five deterministic clock synchronization algorithms are given in Table 5. The *Failures* column gives the failure modes assumed for every component[3]. Column *Structure* indicates the structure of the algorithm. Column *Synchronization event detection* gives the method used for detecting rounds. Column *Clock correction* indicates the clock correction technique used. Finally, column *Specific parameters* gives the value of parameters that are specific to a given building block.

Name	**Failures**			**Structure**	**Synchro. Event Detection**	**Remote Clock Estimation**	**Clock Correction**	**Specific Parameters**
	Cl	Li	Pr					
[ST87]	B	R	B	SYM - FLOOD	MSG	Empty	NAV	$\alpha = 8.5ms$
[LL88]	B	R	B	SYM - FLOOD	SYNC	TT	AV - f_{ftm}	$\beta = 0.1s$
[GZ89]	T	P	C	ASYM - MAST	not required	RCR	AV - f_{fc}	$\varpi = 20ms$
MsgRcrFTM	B	R	B	SYM - FLOOD	MSG	TT	AV - f_{ftm}	
[PB95]	B	R	B	SYM - FLOOD	SYNC	TT	AV - f^{det}_{mean}	$\beta = 0.1s$ $w = 58ms$

Table 5: Selected algorithms

For algorithms using technique SYNC to implement the *resynchronization event detection block*, we take the value $0.1s$ for β parameter, that is a value meeting the equation: $2(1+\rho)(\beta+\epsilon)+(1+\rho)\ max(\delta,\beta+\epsilon)+\rho\delta < R \leq \beta/(4\rho)-\epsilon/\rho-\rho(\beta+\delta+\epsilon)-2\beta-\delta-2\epsilon$ [LL88]. For the algorithm described in [ST87], parameter α is set to value $8.5ms$ to ensure that clocks are never set backwards. The window size for the algorithm given in [PB95] is set

[2]The MsgRcrFTM algorithm has been designed for the purpose of our performance evaluation to show a combination of three building blocks that does not exist in any clock synchronization algorithms we are aware of.

[3]Cl stands for for Clock, Li for Link, and Pr for Processor; R stands for Reliable, C for Crash, P for Performance, B for Byzantine, and T for Timing.

to $58ms$, that is the value minimizing its precision $(\gamma + \delta + \epsilon)$; the adopted convergence function selects the first window containing the largest set of clock estimates, and returns the midpoint of the clocks contained in the window. Parameter ϖ of convergence function f_{fc} used in [GZ89] is set to value $20ms$. In addition, as we do not consider the reintegration of faulty elements, we assume that in [GZ89] the master processor and its clock never crash.

5.2 Simulation results

This section is devoted to an analysis of the behavior of fault-tolerant clock synchronization algorithms, first considering a failure-free environment (Section 5.2.1), then considering the occurrence of failures, both when the number and type of failures respect the fault assumptions made by the algorithm (Section 5.2.2) and when fault assumptions are exceeded (Section 5.2.3).

5.2.1 Results in a failure-free environment

This section is devoted to an analysis of the algorithms *tightness* (i.e., the maximum difference between any two clocks). By definition, at any time, we have for all correct clocks $tightness < \gamma$, where γ is the algorithm precision. Results are obtained through a 100 rounds simulation of every algorithm of Table 5. Figure 2 gives for every algorithm the evolution of the algorithm *tightness* in function of real-time for the first 10 rounds, while Table 6 gives for every algorithm the average and maximum sought tightness for 100 rounds.

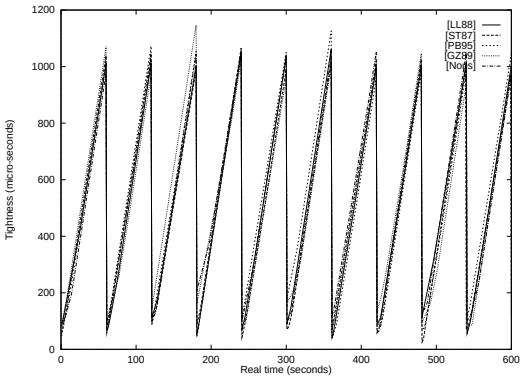

Figure 2: Compared tightness of clock synchronization algorithms

Let us notice that the algorithms tightness is much lower than their precision; for instance, for algorithm [LL88], we have $\gamma = 2.9ms$, which is always much greater than the

Name	Average	Maximum
[ST87]	508.8	1073.2
[LL88]	527.4	1069.2
[PB95]	537.7	1126.4
[GZ89] (Tempo)	535.1	1147.2
MsgRcrFTM	539.7	1128.8

Table 6: Average and maximum tightnesses of clock synchronization algorithms (μs)

algorithm tightness given in Table 6. This comes from the fact that the precision of every algorithm is computed for the worst case situation.

5.2.2 Results in a failure-prone environment

This section is devoted to an analysis of the algorithms behavior in the presence of every kind of failure identified in Section 5.1.2, in the case where the algorithms fault assumptions are verified (i.e., assumption 4 holds). Results are obtained through the simulation of five rounds of every algorithm.

Influence of clock and processor crashes: We have analyzed the behavior of the algorithms described in Table 5 in the presence of a single clock crash (or equivalently a single processor crash) during the simulation duration. As expected, all algorithms support a single clock crash, and the presence of a clock crash only influences the algorithm tightness. Obviously, the tightness changes when the clock having the lowest or largest clock value crashes. Moreover, a clock crash may have an impact on the executions of the *clock correction block*. In algorithms using AV techniques, this situation arises when the crashed clock would have been used by the convergence function if it had not crashed. In algorithms using NAV techniques, it arises when the crashed clock would have influenced the instant when the other clocks resynchronize (i.e., the failed clock was one of the clocks with the $2f+1$ greatest values before the crash).

Figure 3 illustrates the influence on a clock crash on the tightness of [PB95]. The left graph of the figure shows for every clock C_i the value of $C_i(t) - t$ in function of real-time t; the graph to the right of the figure shows the corresponding tightness (the crashed clock is not considered in the tightness computation). At time 54 second, clock 0 crashes, and the tightness decreases as clock 0 was the clock with the largest value. In addition, the crash of clock 0 influences subsequent clock corrections, as clock 0 was belonging before its crash to the window used in [PB95] to compute the correction term A_{p_i}.

Influence of link omissions and link performance failures: We have observed that the impact of link omissions and link performance failures on the algorithms behavior is

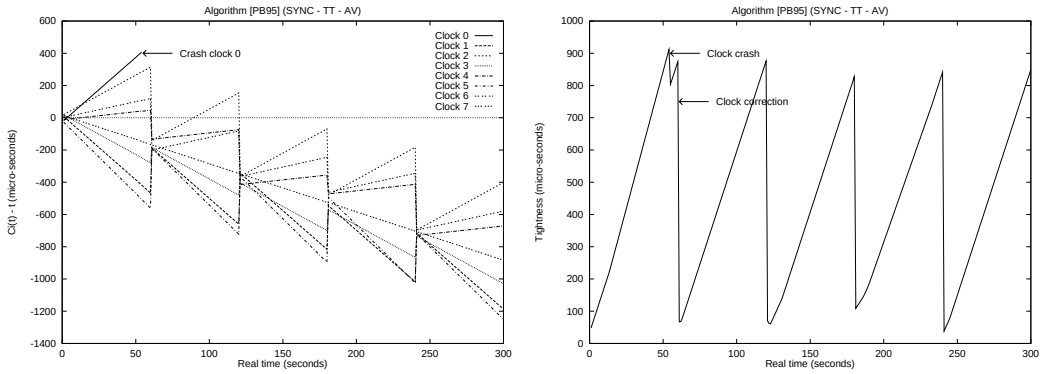

Figure 3: Influence of clock and processor crashes on the tightness of [PB95]

identical to the one of clock and processor crashes. For link omissions, this comes from the fact that omissions can be considered as *transient processor crashes*. For performance failures, very late messages are considered as link omissions by all *remote clock estimation blocks*, thanks to either timeouts or a waiting of a maximum number of messages. Less delayed messages are automatically discarded by every *clock correction block*.

Influence of clock timing failures: The impact of clock timing failures — in the sense that a clock runs too fast — is twofold. First, a clock timing failure may have an influence on the *resynchronization event detection block* by putting forward the time when processors resynchronize. This occurs in [LL88], as well as in all clock synchronization algorithms using the SYNC detection method, in the case where the quickest processor, that is the one with the faulty clock, triggers its resynchronization round sooner than the other processors. Second, it may have an influence on the *remote clock estimation block* by shortening the duration of the estimation interval for the processor on which the faulty clock runs (in case where the end of the estimation interval is detected thanks to a timeout).

However, we have observed during the experiments that the tightness of all the algorithms (computed using every clock value including the faulty one) is always lower than the algorithms precision. This is due to the typical values of ρ, β and R, which makes the two impacts mentioned above scarcely noticeable. For instance, with the simulation parameters used in our experiments, the resynchronization period may be shortened by no more than $3\rho R = 1.8ms$, and the duration of the estimation block may be shortened by no more than $2\rho(\beta + \delta + \epsilon) = 3.2\mu s$.

Influence of clock Byzantine failures: We have analyzed the behavior of the algorithms described in Table 5 in the presence of a single clock affected by a Byzantine failure. As

expected, all algorithms excepted [GZ89] support a single Byzantine failure, i.e., all *correct* clocks differ by at most the algorithm precision γ. However, clock synchronization algorithms differ by their ability to reintegrate Byzantine clocks depending on the method they use to implement their *resynchronization event detection block* and their *clock correction block*. Algorithms using technique MSG for detecting resynchronization events, and a NAV clock correction technique are able to reintegrate Byzantine clocks while algorithms using SYNC are not.

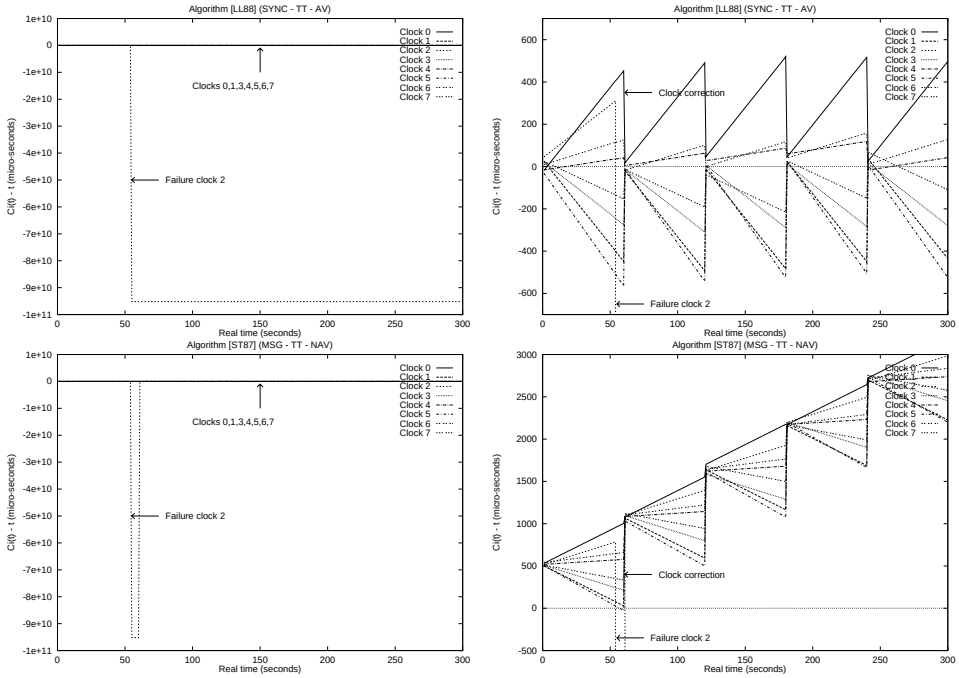

Figure 4: Influence of Byzantine failures on [LL88] and [ST87]

Figure 4 illustrates this difference on algorithms [LL88] and [ST87]. In this figure, four graphs are shown. The two upper ones (resp. the two lower ones) only differ by their scale in y. The y scale of the left graphs is multiplied by 10^6 to emphasize the behavior of the Byzantine clock. On these graphs, clock 2 suffers from a Byzantine failure at real-time $54s$. In algorithm [LL88] (as well as in any other algorithm using a SYNC *resynchronization event detection block*), the faulty clock does not resynchronize anymore with the others; it receives estimation messages but stops starting new rounds due to its erroneous clock value. In contrast, in algorithm [ST87] (as well as in any other algorithm using the MSG *resynchronization event detection block*) the faulty clock continues to start new rounds. In

addition, as [ST87] *clock correction block* uses a NAV technique, which computes clock correction independently of the value of remote clock estimates, the Byzantine clock is reintegrated.

5.2.3 Results when fault assumptions are not verified

We analyze in this paragraph the influence of unexpected failures on the algorithms behavior. Results are obtained through the simulation of five rounds of every algorithm.

Behavior in the presence of a single unexpected processor crash: Our first experiment concerns the behavior of clock synchronization algorithms when a single clock crash occurs while the algorithm is designed to work in a reliable environment (i.e., simulation parameter f is equal to 0). We have observed that all algorithms behave correctly, i.e., the tightness of correct clocks remains lower than the algorithm precision γ. Indeed, whatever *resynchronization event detection block* is used, a clock crash does not prevent correct clocks to detect rounds. Similarly, whatever *remote clock estimation block* is used, a crashed clock does not prevent correct clocks to estimate the value of every other correct clock. Finally, as to our knowledge every *clock correction block* does not consider crashed clocks, an unexpected clock crash does not cause correct clocks to lose synchronization, as far as enough correct clocks exist to be able to apply the clock correction technique.

Simulations of a single network failure per round (link omission or link performance failure) have been performed. The obtained results are identical to the ones obtained when simulating crash failures.

Behavior in the presence of a single unexpected Byzantine failure: Our second experiment concerns the behavior of clock synchronization algorithms when one clock Byzantine failure occurs while the algorithm is designed to operate in a fault-free environment. Simulations show that their behavior differ depending on their *resynchronization event detection block*. Algorithms using the SYNC *resynchronization event detection block* continue to behave correctly, i.e., the tightness of correct clocks remains lower than the algorithm precision γ. This is due to the fact that the faulty clock does not detect synchronization events anymore, and thus stops participating to the clock synchronization algorithm.

In contrast, in algorithms using the MSG *resynchronization event detection block*, all clocks (including the faulty one) continue to detect new rounds. The behavior of such algorithms then depends on their *clock correction block*. In AV *clock correction blocks*, the Byzantine clock can prevent correct clocks from remaining synchronized if it is used by the convergence function. For instance, the f_{ftm} convergence function used in algorithm MsgRcrFTM (see left graph of figure 5) uses the value of the Byzantine clock (clock 3 in the figure) to compute the adjustment term; correct clocks then become corrupted by

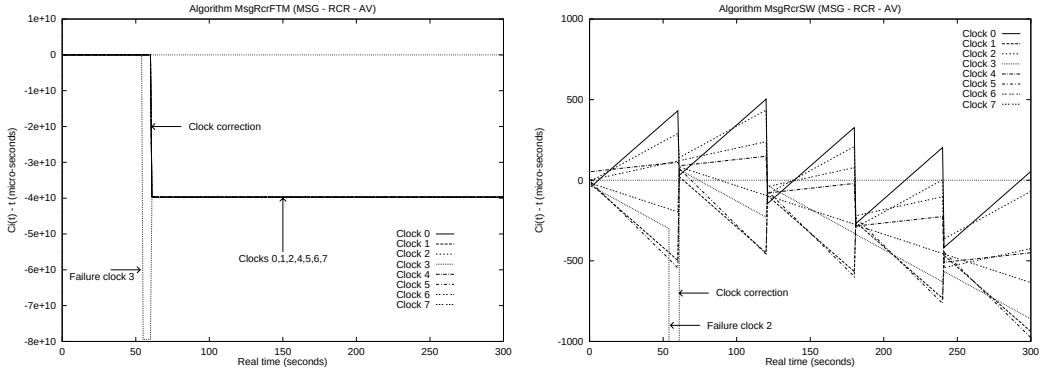

Figure 5: Influence of a single unexpected Byzantine failure on MsgRcrFTM and MsgRcrSW

the Byzantine clock. If the f_{sw} convergence function is used instead of f_{ftm} (see right graph of figure 5), the Byzantine clock is not used by the convergence function. Correct clocks can then remain synchronized with each others. In addition, due to the use of the MSG *resynchronization event detection block* the Byzantine clock (clock 2 in the figure) is reintegrated.

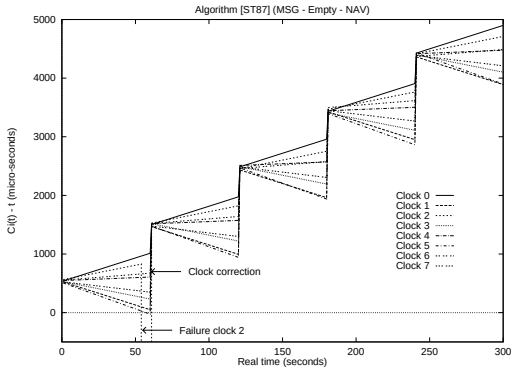

Figure 6: Influence of a single unexpected Byzantine failure on [ST87]

When a NAV *clock correction blocks* is used (see figure 6), as NAV never uses the value of remote clock estimates to compute clock adjustments, correct clocks remain synchronized. Moreover, for the same reasons as below, the Byzantine clock (clock 2 in the figure) is reintegrated.

6 Conclusion

This paper was devoted to a survey of software clock synchronization algorithms. A wide panel of algorithms was classified thanks to a taxonomy taking into account the system model, failure model, structure of the algorithm and three basic building blocks we have identified. While deterministic, probabilistic and statistical clock synchronization algorithms use the same building blocks (see table 1), the main difference between these three classes of algorithms concerns their respective objectives. Deterministic algorithms aim at guaranteeing a worst-case precision. Probabilistic algorithms take advantage of the current working conditions, by invoking successive round-trip exchanges, to reach a tight (but not guaranteed) precision. Statistical algorithms strongly rely on their knowledge of delay distributions to reduce the number of synchronization messages to get a tight (but not guaranteed) precision at the expense of loosing synchronization without any mean to detect it.

A performance evaluation of a panel of deterministic clock synchronization algorithms [LL88, ST87, PB95, GZ89] has been presented, and has shown that our taxonomy allows to classify a large range of clock synchronization algorithms. The performance evaluation also allowed us to analyze the behavior algorithms [LL88, ST87, PB95, GZ89] in the presence of various kinds of failures (crash, omission, timing, performance, and Byzantine), both when the number and type of failures respect the fault assumptions made by the algorithm and when fault assumptions are exceeded. Results of this evaluation are very satisfying. Indeed, beyond the good behavior of these algorithms in presence of failures whose kind and number meet the fault assumptions made by the algorithm, their behavior face to unexpected failures is quite surprising. None of these algorithms is affected by a large number of failures, which make them very robust in an aggressive environment. The only exception concerns algorithms based on averaging *clock correction blocks* whose behavior can be contaminated by the presence of too many clock Byzantine failures. In such a situation, Byzantine clocks can prevent correct clocks from remaining synchronized.

Acknowledgments

We are grateful to Flaviu Cristian for pointing out the difference between probabilistic and statistical clock synchronization algorithms, and for his helpful comments on initial drafts of this paper.

7 Glossary

n	total number of processors
p_i, p_j	processor identifiers
ρ	maximum rate of drift of the physical clocks, in clock seconds per real seconds
ν	maximum rate of drift of the logical clocks L_p, in clock seconds per real seconds
$H_{p_i}(t)$	physical clock of processor p
$A_{p_i}(t)$	correction factor at real time t
$L_{p_i}(t)$	processor p's logical time at real time t ($L_p(t) = H_p(t) + A_p(t)$)
β	maximum difference in real-time between any two non faulty processors at the beginning of the clock synchronization algorithm
γ	upper bound on closeness of logical time
φ	maximum difference between logical time and real time
t, t_1, t_2	real times
T, T_1, T_2	clock times
π, ϖ	constant values
Δ	maximum difference between a correct reference clock and real time
Σ	bound of the correction term used at each resynchronization
R	duration of a round, measured in logical clock time units
Θ	maximum reading error of a remote clock
Γ_t	maximum interval between receipt of an uncorrupted message in broadcast networks
Γ_a	maximum duration of an agreement protocol
δ	midpoint of range of possible message delays
ϵ	uncertainty in message delays
$\overline{\delta}$	expectation of the message delays ($\overline{\delta} \in [\delta - \epsilon, \delta + \epsilon]$)
f	maximum number of faulty elements

References

[Arv94] K. Arvind. Probabilistic clock synchronization in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):474–487, May 1994.

[CAS86] F. Cristian, H. Aghili, and R. Strong. Clock synchronization in the presence of omission and performance failures, and processor joins. In *Proc. of 16th International Symposium on Fault-Tolerant Computing Systems*, July 1986.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcsat: from simple message diffusion to byzantine agreement. In *Proc. of 15th International Symposium on Fault-Tolerant Computing Systems*, 1985.

[Cri89] F. Cristian. Probabilistic clock synchronization. In *Distributed Computing*, volume 3, pages 146–158. Springer Verlag, 1989.

[DHSS95] D. Dolev, J. Y. Halpern, B. Simons, and R. Strong. Dynamic fault-tolerant clock synchronization. *Journal of the ACM*, 42(1):143–185, January 1995.

[FC95a] C. Fetzer and F. Cristian. Lower bounds for function based clock synchronization. In *Proc. of 14th International Symposium on Principles of Distributed Computing*, August 1995.

[FC95b] C. Fetzer and F. Cristian. An optimal internal clock synchronization algorithm. In *Proc. of the 10th Annual IEEE Conference on Computer Assurance*, June 1995.

[FC97] C. Fetzer and F. Cristian. Integrating external and internal clock synchronization. *Journal of Real-Time Systems*, 12(2):123–172, 1997.

[GZ86] R. Gusella and S. Zatti. An election algorithm for a distributed clock synchronization program. In *Proc. of 6th International Conference on Distributed Computing Systems*, pages 364–373, 1986.

[GZ89] R. Gusella and S. Zatti. The accuracy of the clock synchronization achieved by TEMPO in berkeley UNIX 4.3BSD. *IEEE Transactions on Software Engineering*, 15(7):847–853, July 1989.

[HSSD84] J. Y. Halpern, H. R. Strong, B. B. Simons, and D. Dolev. Fault-tolerant clock synchronization. In *Proc. of 3rd International Symposium on Principles of Distributed Computing*, pages 89–102, 1984.

[KO87] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time computer systems. *IEEE Transactions on Computers*, C-36(8):933–940, August 1987.

[Lam78] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LL88] J. Lundelius and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77:1–36, 1988.

[LMS85] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, July 1985.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *acmtpls*, 4:383–401, 1982.

[Mar83] K. Marzullo. *Loosely-Coupled Distributed Services: A Distributed Time Service.* PhD thesis, Stanford University, Computer Systems Laboratory, 1983.

[Mil91] D. L. Mills. Internet time synchronization: The network time protocol. *IEEE Transactions on Computers*, 39(10):1482–1493, October 1991.

[MO83] K. Marzullo and S. Owicki. Maintaining the time in a distributed system. In *Proc. of 2nd International Symposium on Principles of Distributed Computing*, pages 295–305, 1983.

[MS85] S. R. Mahaney and F. B. Schneider. Inexact agreement: Accuracy, precision and graceful degradation. In *Proc. of 4th International Symposium on Principles of Distributed Computing*, pages 237–249, August 1985.

[OS94] A. Olson and K. G. Shin. Fault-tolerant clock synchronization in large multicomputer systems. *IEEE Transactions on Parallel and Distributed Systems*, 5, 1994.

[PB95] M. J. Pfluegl and D. M. Blough. A new and improved algorithm for fault-tolerant clock synchronization. *Journal of Parallel and Distributed Computing*, 27:1–14, 1995.

[RSB90] P. Ramanathan, K. G. Shin, and R. W. Butler. Fault-tolerant clock synchronization in distributed systems. *IEEE Computer*, 23(10):33–44, October 1990.

[SC90] F. Schmuck and F. Cristian. Continuous clock amortization need not affect the precision of a clock synchronization algorithm. In *Proc. of 9th International Symposium on Principles of Distributed Computing*, pages 133–144, 1990.

[Sch86] F. B. Schneider. A paradigm for reliable clock synchronization. Technical Report TR86-735, Computer Science Department, Cornell University, February 1986.

[Sch92] H. Schwetman. Csim reference manual (revision 16). Microelectronic and computer technology corporation, Austin TX, june 1992.

[SR87] K. G. Shin and R. Ramanathan. Clock synchronization of a large multiprocessor system in the presence of malicious faults. *IEEE Transactions on Computers*, C-36(1):2–12, 1987.

[ST87] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[SWL90] B. Simons, J. Lundelius Welch, and N. Lynch. An overview of clock synchronization. In Spector, editor, *Asilomar Workshop on Fault-tolerant Distributed Computing Conference*, volume 448, pages 84–96. Lecture Notes in Computer Science, 1990.

[TFKW89] P. Thambidurai, A. Finn, A. Kieckhafer, and C. Walter. Clock synchronization in MAFT. In *Proc. of 19th International Symposium on Fault-Tolerant Computing Systems*, pages 142–149, 1989.

[VCR97] P. Verissimo, A. Casimiro, and L. Rodrigues. Cesiumspray: a precise and accurate global time service for large scale systems. *Journal of Real-Time Systems*, 12:243–294, 1997.

[VR92] P. Verissimo and L. Rodrigues. A posteriori agreement for fault-tolerant clock synchronization on broadcast networks. In *Proc. of 22nd International Symposium on Fault-Tolerant Computing Systems*, 1992.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399