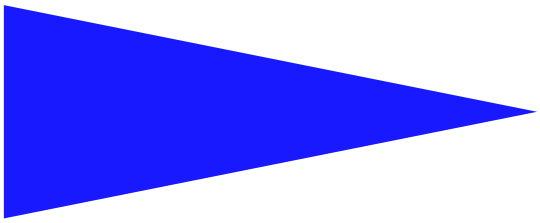
I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1103



A TAXONOMY OF CLOCK SYNCHRONIZATION ALGORITHMS

EMMANUELLE ANCEAUME AND ISABELLE PUAUT

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

A Taxonomy of Clock Synchronization Algorithms

Emmanuelle Anceaume and Isabelle Puaut

Thème 1 — Réseaux et systèmes
Projet Solidor

Publication interne n ° 1103 — Juillet 1997 — 25 pages

Abstract: Clock synchronization algorithms ensure that physically dispersed processors have a common knowledge of time. This paper proposes a taxonomy adapted to all published software fault-tolerant clock synchronization algorithms: deterministic and probabilistic, internal and external, and resilient from crash to Byzantine failures. We classify clock synchronization algorithms according to their internal structure and to three othogonal and independent basic building blocks. Our taxonomy 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, distributed systems, real-time systems, deterministic, probabilistic

(Résumé : tsvp)

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

Centre National de la Recherche Scientifique
(UPRESSA 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Une Classification des 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 une classification adaptée à tous les algorithmes de synchronisation d'horloges évoluant dans un environnement sujet aux défaillances, qu'ils soient déterministes ou probabilistes, et qu'ils assurent une synchronisation interne ou externe. Les algorithmes de synchronisation d'horloges étudiés sont classés selon leur structure interne et selon trois blocs de base indépendants. Cette classification est conçue pour guider le concepteur d'un algorithme de synchronisation d'horloges dans le choix de l'algorithme le mieux adapté 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. Par ailleurs, la classification proposée utilise une notation uniforme, qui permet de comparer les algorithmes existants selon le modèle de fautes qu'il supportent, les blocs de base qu'ils utilisent, les propriétés qu'ils assurent et le coût associé.

Mots clés : Synchronisation d'horloges, systèmes distribués, systèmes temps-réel, déterministe, probabiliste

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. Finally, the most important difficulty is to support faulty elements, which are common in distributed systems.

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. 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. In contrast, software clock synchronization algorithms [CAS86, HSSD84, ST87, LL88, LMS85, VCR97, PB95, GZ89, MS85, FC95a, FC97, 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 about maximum message delays. While probabilistic algorithms do not assume anything on delay distributions, statistical algorithms assume that the expectation and standard deviation of the delay distributions are known. In contrast to deterministic clock synchronization algorithms, probabilistic and statistical algorithms guarantee a constant maximum deviation between any clock values with a probability strictly less than one. In probabilistic algorithms, a clock knows at any time if it is synchronized or not with the others, whereas in statistical algorithms, clocks do not know how far apart they are from each others.

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 (i.e., 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 fault-tolerant 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 all algorithms are classified according to a larger set of parameters: system synchrony, failure model, structure of the algorithm and basic building blocks. Our taxonomy 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. In addition, we use 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 and their cost in terms of number of messages exchanged. 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, for space considerations, we do not detail issues specific to clock synchronization in large scale distributed systems (e.g. [SR87, VCR97]).

We classify clock synchronization algorithms according to their internal structure (symmetry of the algorithm) and to three orthogonal and independent 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*. Each building block in our taxonomy is illustrated with algorithms relying on it.

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. The proposed classification of clock synchronization algorithms is introduced in Section 4. Finally, a summary of the characteristics of the most referenced algorithms is proposed in Section 5.

2 System model

The study of any clock synchronization algorithm requires a precise 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 mode 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 vs 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, they introduce *logical clocks*. 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 either be a discrete value changed at each resynchronization [SC90, ST87] or a linear function of time [SC90, Cri89].

In order to implement *external* synchronization algorithms, 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*
$| H_{p_r}(t) - t | \leq \Delta$

2.2 Network synchrony

In distributed 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. send, transport and receive) a message exist. This 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 $[\delta - \epsilon, \delta + \epsilon]$.*

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 algorithm *precision*). 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 logical clocks can only be guaranteed with some non-null probability.

2.3 Failure mode

Failures can be defined as deviations 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). Other ones, such as [GZ89] assume timing failures. , a more restricted failure mode prohibiting conflicting information.

Clock timing failure: [CASD85] *A local hardware clock commits a timing failure if it does not meet assumption 1, i.e., is not* ρ *– bounded.*

Clock Byzantine failure: [LSP82] *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.*

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, OS94], commit performance failures as in [VCR97, CAS86, Cri89] or more generally, commit Byzantine failures as in [HSSD84, ST87, LL88, LMS85, PB95, MS85, FC95a, FC97].

Processor crash failure: [CASD85] *A processor commits a crash failure if it behaves correctly and then stops executing forever (permanent failure).*

Processor performance failure: [CASD85] *A processor commits a performance failure if it completes a computation step in more than the specified value.*

Processor Byzantine failure: [LSP82] *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 (the network never crashes).

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. Obviously, 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 it, 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, communication links), as stated in assumption 4.

Assumption 4 (Bounded number of faulty components) *There are at most f faulty components in the system during the execution of the clock synchronization algorithm, with f a positive integer*[1].

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, [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.*

[1] The value of f depends on the failure mode assumed. For instance, to tolerate f processor Byzantine failures without authentification, f must satisfy $f \leq \frac{n}{3}$, with n the number of processors [LL84].

Another way to rule out trivial solutions is to require that logical clocks are permanently within a narrow envelope of real time (property 3).

Property 3 (Accuracy) *For any correct processor $p_i \in \mathcal{P}$, and for any 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], and is termed *bounded drift rate* in [FC97]. The best accuracy ν 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, $\mid L_{p_i}(t) - t \mid \leq \varphi$.*

Notice that some clock synchronization algorithms may impose 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

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, whereas 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. Notice that in [MS85, FC95a, FC97], 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 clock synchronization messages to every other one, and uses the messages received from the remote processors to correct its clock. Symmetric algorithms can be split into two classes, *flooding-based* and *ring-based*, depending on the virtual path taken for transmitting a message from every processor to every other one.

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, as they do not exhibit a single point of failure. However, they may require up to n^2 messages to disseminate a clock synchronization message to all processors in $\mathcal{P}$. This large number of messages can be lowered to n if a broadcast network is used [VCR97].

The virtual ring scheme was proposed in [OS94] to decrease the number of exchanged messages experienced in flooding-based schemes. In the ring scheme, all processors in $\mathcal{P}$ 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 along the ring, each processor adds its own data to the message. Compared with flooding-based schemes, virtual ring schemes need to exchange a smaller number of messages (only n messages per resynchronization are used), but need extensions in order to support processor and network failures.

4.1.2 Asymmetric schemes

In asymmetric schemes, also called *master-slave* schemes, processors involved in the clock synchronization algorithm play different roles. One processor in $\mathcal{P}$ is designed as the *master*, and the other processors are designed as the *slaves*. Within this scheme, two variants exist. In the former one, called *master-controlled* scheme [GZ89], 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. In the latter variant, commonly called *slave-controlled* scheme, the master acts as a reference processor providing time with respect to which all the slaves resynchronize [Cri89, Arv94].

The obvious advantage of asymmetric schemes is their low cost in terms 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, such as fault detection or election of a new master to be avoided. Another drawback of the asymmetric scheme is the presence of a single master that can be swamped by a large number of synchronization messages, thus invalidating in some way the assumptions made on the communication delays.

4.2 Clock synchronization building blocks

In any clock synchronization algorithm — 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 logical clock according to the result of the second step. The *clock correction block* (Section 4.2.3) 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 Resynchronization event detection block

The objective of p_i's *resynchronization event detection block* is to trigger a *resynchronization event* informing processor p_i to start the clock synchronization algorithm. Indeed, due to clock drifts, clocks must be re-synchronized often enough to guarantee the agreement property. A common way to resynchronize clocks is to periodically trigger the clock syn-

chronization 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] 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 paragraphs.

Detection technique based on initially synchronized clocks

The mostly used technique is to assume initially 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, and k is an integer ≥ 1.

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], we must have $2(1+\rho)(\beta+\epsilon)+(1+\rho)\ max(\delta,\beta+\epsilon)+\rho\delta < R \leq \beta/(4\rho)-\epsilon/\rho-\rho(\beta+\delta+\epsilon)-2\beta-\delta-2\epsilon$.

Detection technique based on message exchanges

As mentioned in [LMS85], the precision γ achieved by algorithms based on $\beta-$synchronized clocks is very sensitive to the initial clock synchronization value β. Another way to detect the beginning of a round consists for a processor in sending a message to all processors in $\mathcal{P}$ as soon as its logical clock reaches the predefined value kR. Upon receipt of such a message, processor $p_i \in \mathcal{P}$ starts a new clock synchronization round. 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 [VCR97].

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 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 p_i's *remote clock estimation* block is a *clock estimation set* containing the set of remote clock estimates, and taken as input of the local *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 that every remote clock can be estimated with a small bounded error Θ, the error obviously depending on the actual clock estimation technique used.

The time transmission (TT) technique

In the *time transmission* (TT) technique, processor p_i sends its local clock in a message. The receiving processor p_j uses the information carried in this message to estimate p_i's clock. Depending on whether communication delays are bounded or not (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], p_j sends its message to all processes in $\mathcal{P}$ at a fixed local logical time, say T. Meanwhile, p_j collects messages from the other processors within a particular amount of time measured on its logical clock[2]. Processor p_j stores the local time at which these messages are received in its *clock estimation set*. The receipt time of p_i's message is used as an estimate of p_i's clock. 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 p_i is non faulty. Length of this interval equals $(1+\rho)(\delta+\epsilon+\beta)$. At receipt time, p_i's clock belongs to the interval $[T+(1-\rho)(\delta-\epsilon-\beta), T+(1+\rho)(\delta+\epsilon+\beta)]$, thus obtaining a maximum estimation error of $2(\epsilon+\beta+\rho\delta)$.

In case where communication delays are not bounded, a statistical variant of TT has been identified [OS94, Arv94]. In this variant, the absence of both initially synchronized clocks and precise upper bounds on the transmission delays are overcome by s successive transmissions of timestamped synchronization messages. Note that this variant 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

[2] According to the structure of the algorithm (symmetric, slave-controlled asymmetric or master-cpntrolled asymmetr ic) and to the failure mode assumed, the number of messages p_j must wait 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 deal with the case where the upper bound of communication delays is unknown. The *RCR* technique works as follows. 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. Processor p_j stores T in its *clock estimation set*. Value T is used as an estimate of p_i's clock. 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 p_i's message is $[T + (\delta - \epsilon)(1 - \rho), T + 2D(1 + 2\rho) - (\delta - \epsilon)(1 + \rho)]$, thus obtaining a maximum estimation error of $2D(1 + 2\rho) - 2(\delta - \epsilon)$.

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 estimation error. If p_i wants to have an estimation 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 $U = (1 - 2\rho)(\Theta + \delta - \epsilon)$.

Benefits of the *RCR* clock estimation technique, used in [Cri89, GZ89, OS94, VCR97], 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 logical clock in relation with the other ones. The *clock correction* block of processor p_i takes as input p_i's *clock estimation set*. This block has no output as its effect is the correction of p_i's logical clock. The adjustment of a logical clock can be done by using either the clock estimates contained in the *clock estimation set* or the fact that they have been received. In the former case, p_i's adjustment A_{p_i} is obtained by applying a convergence function to the *clock estimation set*

(*convergence-averaging* techniques), while in the latter case (*convergence-nonaveraging* techniques) A_{p_i} is a value that does not depend on the values contained in the *clock estimation set*.

Convergence-averaging techniques

Convergence-averaging techniques, also called *convergence function based* techniques, use a so-called *convergence function*, which takes as arguments the clock estimates contained in the *clock estimation set* and returns some kind of averaging on these estimates. An exhaustive list of convergence functions introduced before 1987, is given in [Sch86].

In the following, $f(p_i, x_1, \cdots, x_n)$ identifies a convergence function. Argument p_i is the processor requesting clock synchronization, and $x_1, \cdots, x_n$ are the estimated clock values contained in p_i's *clock estimation set*.

The way the adjustment term A_{p_i} (see Section 2) is computed differs according to the algorithms. For instance in [LL88], A_{p_i} is set to $T + \delta - f_{ftm}(p_i, x_1, \cdots, x_n)$, where T is the time at which the algorithm is started and $f_{ftm}(p_i, x_1, \cdots, x_n)$ is the result of the convergence function $f_{ftm}(p_i, x_1, \cdots, x_n)$ applied to the elements of p_i's *clock estimation set*.

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

- *Interactive convergence function.* This function, also known as the *Egocentric average* function f_e is used in [LMS85]. $f_e(p_i, x_1, \cdots, x_n)$ returns the average of all arguments x_1 through x_n, where x_j with $1 \leq j \leq n$ is kept intact if it is no more than ϖ from x_i and is replaced by x_i 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 convergence function f_e is that no sorting algorithm must be applied to reject faulty clock values.

- *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 significant.

- *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. f_{ftm} is based on the assumption that faulty clocks run too fast or too slow, and thus that good clocks will generally lie between.

- *Differential fault-tolerant midpoint function.* This function denoted f_{dftm}, has been introduced in [FC95b] and used in [FC97]. 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:

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

  where T is p_i's logical time and Θ is the maximum reading error of a distant clock, which has been estimated in [Cri89] to value $2\rho\delta + 2\epsilon(1+\rho)$ when RCR is used.

- *Sliding window function.* This function, proposed in [PB95], selects a fixed size window 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 $[L_i(t) - e_i(t), L_i(t) + e_i(t)]$, where $e_i(t)$ is the maximum error on p_i's clock estimate, and returns an interval for the corrected clock value. More precisely:

$$S_i = \{x_j \mid \mathit{consistent}(p_i, p_j)\}$$
$$f_{mm}(p_i, x_1, \cdots, x_n) = [L_m(t) - e_m(t), L_m(t) + e_m(t)], \mathit{where}\ \forall i \in S_i, e_m(t) \leq e_i(t)$$

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

- *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, 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

Unlike convergence-averaging techniques which compute a new clock value by using the contents of the *clock estimation set*, convergence-nonaveraging techniques only use the fact that a predefined number of remote clocks estimates have been received. The number of expected clock estimates depends on the type and the number of tolerated failures. For instance, the clock correction block requires that the *clock estimation set* contains $2f + 1$ estimates to support f Byzantine failures with no authentification, or $f + 1$ estimates in case where authentification is used, or one single estimate in case of performance failures [CAS86].

The clock correction block updates the logical clock with a value that differs according to the algorithms. The logical clock is set to $kR + \pi$ in [ST87], to kR in [CAS86, HSSD84], where k is the round number, R is the round duration and π is a constant. In [VCR97], the logical clock is set to the value of one of the clock estimates contained in the *clock estimation set*.

5 Illustration of the taxonomy and conclusion

This section illustrates our taxonomy on a long (but not exhaustive) list of the most referenced deterministic, probabilistic and statistical clock synchronization algorithms. By lack of space, we cannot afford to present a full description of each algorithm according to our taxonomy. However, we present in Section 5.1 their structure and building blocks and in Section 5.2, their performance and cost.

5.1 Classification of clock synchronization algorithms

Table 1 presents the classification of the most referenced clock synchronization algorithms. This table shows that all the clock synchronization algorithms, whatever their type, can be

completely described by means of the three identified building blocks 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.

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	TT	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	NAV
[PB95]	D	B	R	B	SYM - FLOOD	SYNC	TT	AV - f_{sw}
[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 labelled 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)

5.2 Performance of clock synchronization algorithms

Table 2 gives for each algorithm 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.

Results are given in two tables. Table 2 is devoted to performance of deterministic algorithms, while table 3 concerns probabilistic ones. Note that in case of probabilistic clock synchronization algorithms, accuracy is never specified by the authors, and thus is not reported in table 3.

5.3 Concluding remarks

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 simulation study is under way and is analysing the performance and validity assumptions of each building block, as well as the compatibility among themselves.

Reference	Type	Redundancy	Precision[a]	Accuracy	Msg.
[CAS86] [b]	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] [c]	I	$3f+1$	$5\epsilon+4\rho\delta+4\rho R$	$\rho+\frac{\epsilon}{R_{min}}$	n^2
[LMS85] (CSM) [d]	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] [e]	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)[f]	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] [g]	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) [h]	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: Performance of deterministic clock synchronization algorithms

[a] The drift of hardware clocks (ρ) being very small, we make in the commonly adopted assumption that the terms ρ^m for $m \geq 2$ can be neglected.

[b] 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].

[c] 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).

[d] 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).

[e] 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 algorithm takes advantage of a broadcast network, in which nodes receiving an uncorrupted message receive it at real time values that differs at most by a known small constant Γ_t. Value Γ_a is the maximum duration of an agreement protocol.

[f] The algorithm described in [GZ89] does not inherently support the failure of the master process. When it 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 $\delta+\epsilon$.

[g] 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 does not drift.

[h] The basic requirement of the algorithm is that clocks are represented validity intervals. The algorithms are extended to support faulty clocks in [Mar83]. $E_m(t)$ 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 mentionned
[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: Performance of probabilistic 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 seeked 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. erf is the error function, and γ 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 echanged.

[c] The value given in the *Precision* column is the best precision achievable when an precision of γ_{synch} is obtained just after resynchronization. p stands for the probability a precision of γ_{synch} is obtained, σ^2 is the standard deviation of the message delay, and $erf^{-1}(p)$ is the inverse of the complementary error function defined as $erfc(u) = 1 - erf(u)$, where $erf(u)$ is by definition equal to $\frac{2}{\sqrt{\pi}} \int_0^u e^{-y^2} dy$.

[d] The value given in the *Precision* column is the best precision achievable assuming that no failure occurs. 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 $2U$.

6 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
$H_p(t)$	physical clock of processor p
$A_p(t)$	correction factor at real time t
$L_p(t)$	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
γ_{synch}	upper bound on closeness of logical time at resynchronization intervals
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

Acknowledgements

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.

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. Halpern, H. Strong, 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.

[LL84] J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2/3):190–204, August 1984.

[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, Standford University, Computer Systems Laboratory, 1983.

[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. Mahaney and F. 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. Shin. Fault-tolerant clock synchronization in large multicomputer systems. *IEEE Transactions on Parallel and Distributed Systems*, 5, 1994.

[PB95] M. Pfluegl and D. 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. Shin, and R. 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. Schneider. A paradigm for reliable clock synchronization. Technical Report TR86-735, Computer Science Department, Cornell University, February 1986.

[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.

[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.