On-line and Post-processing Timestamp Correspondence for Free-Running Clock Nodes, using a Network Clock

Olivier Bezet
VERIMAG UMR UJF/INPG/CNRS 5104
2 av.de Vignate
38610 Gières FRANCE
Olivier.Bezet@imag.fr

Véronique Cherfaoui
Heudiasyc UMR 6599 CNRS
Université de Technologie de Compiègne
60205 Compiègne Cedex, FRANCE
Veronique.Cherfaoui@hds.utc.fr

Abstract - ***Timestamp conversion is an important consideration in the deployment of distributed architectures. In this article we propose a precise, low-cost solution for on-line and post-processing timestamp conversion in distributed architectures, robust as regards the plugging and unplugging of hardware and the addition of new nodes (that is to say the different pieces of hardware connected to the network), not synchronized, and with no negative impact on the conversion quality.***

Each node (e.g., a computer) has at least one free-running clock. This clock's time is the reference for all events used by the node. When the local node needs to record the time of an event timestamped by a remote node, the time is converted from the remote node's time to the local node's time. Interval timestamping is used, to take account of time imperfections (e.g. sensor and computer latencies, or due to time conversion between the different computers).

A network clock is used, enabling a precise conversion and avoiding exchanges of messages for the conversion of clock correspondences. Moreover, it allows an unlimited number of nodes in the network.

Keywords: Distributed Architecture, Timestamping Conversion, Timestamping Error Modeling, Free Running Clock Nodes, Network Clock

1 Introduction

Most data processing algorithms for use in dynamic environments take account of the temporal aspects of data acquisition, that is to say monotony, synchronization and dynamic models. For example, in perception and control applications, the well-known Kalman Filter uses a prediction model and updating equations to combine out-of-sequence data arriving at different times [Bar-Shalom, 2000]. When data acquisition and data processing are centralized in the same computer, its clock becomes the reference time, and temporal data organization is made possible via a timestamping process. Cooperative sensing implies a reference time for all computers linked to the network, determining exactly when the different events occur. Time-sharing techniques can be different depending on the requirements and on the networks.

Implementing a distributed sensor acquisition and processing architecture means paying particular attention to timestamping problems. One solution might be to connect sensors using hard synchronization or/and to use real-time distributed systems. This approach is expensive and precludes application evolution. Another solution is to use a common pool of computers and to take account of delays (the time required for data transmission). Several approximations are used: delays can be ignored or assumed to be constant. Errors can therefore appear when the true delay does not correspond to the assumptions made.

An obvious first step is to set all computer clocks to the same time at system startup. However, clocks do not run at the same speed, since each one is unique at the atomic scale, and has a particular 'drift' which, moreover, varies depending on external conditions (mainly temperature, but also vibrations), on its age and on other parameters. We observed, for example, a maximum drift of 200 $\mu s/s$ for standard desktop PCs, which gives a time variation of 0.72 s for one hour. For precise applications, this drift is by no means negligible.

Two software solutions are traditionally used to precisely share the time between several computers. The first solution is to synchronize all computer clocks (clock synchronization), and the second is to convert a timestamp from one clock reference to another clock reference each time it is necessary (timestamp conversion).

This paper proposes a method for managing data timestamping and accuracy. Timestamping inaccuracy is represented by interval dates. We describe a method for estimating this interval, using standard computers in a distributed architecture, without dedicated systems. The proposed method, which to our knowledge has never previously been used for timestamp conversion, works both on-line and for post-processing, and harnesses the advantages of a network clock for timestamp correspondence computation. Some networks, such as synchronous bus-networks, use a clock to synchronize exchanges of messages. It is a common time accessible to all nodes, but this time is not monotonous, so it cannot be taken as global reference time. We look at how it might be possible to overcome this obstacle and hence obtain all the advantages of a network clock. The proposed date-sharing method is intended for applications where the network is liable to evolve (through the addition or removal of nodes, sensors and networks), or where it is unsafe (for example as a result of dysfunctional switches). It provides a low-cost conversion, and the precision is in the order of three or four times the network clock granularity. The error conversion is estimated, meaning that data integrity is preserved, making it suitable for cases such as sensor data fusion algorithms.

The intended application for our work is in the domain of intelligent vehicles, and particularly in the embedded distributed calculation of driver behavior indicators. Several computers are needed in the car to record raw sensor data and compute all the required indicators. This is a highly dynamic system: a car has a relatively high speed and every single event needs to be recorded, which means that timestamping has to be very robust and precise. Furthermore, good on-line timestamps give rise to good dating estimations and therefore good data fusion results. Distributed architectures can be useful for reducing computer workload, increasing computer reactivity and enhancing real-time behavior. However, distributed architectures are often avoided because of the complexity of their deployment, the cost of the hardware, and the poor precision of the algorithms. Nowadays fast prototyping is more and more widely used, which means that different ideas may be tested quickly; when the best compromise is found, a stable version can be implemented with adapted hardware.

The article is divided into four parts, plus a conclusion. First, related work is listed in Sect. 2. Then, the system model is presented in Sect. 2.2, followed by the proposed approaches in Sect. 3. Theoretical and experimental performances are described in Sect. 4. Finally, conclusions and future work are laid out in Sect. 5.

2 Related work

The subject of this article includes several domains of study. The first is computer clock synchronization and timestamp conversion. The second encompasses those domains inherent in the proposed system model, including interval manipulation, timestamping with intervals, and synchronous networks. The paper proposes a new method related to timestamp conversion, using a synchronous network and using interval timestamping.

2.1 Computer clock synchronization and timestamp conversion

Much work has been devoted to clock synchronization or time conversion. [Kopetz et al., 2006] proposes a classification of the different approaches and describes in detail the different possibilities for clock synchronization. Computer clocks can be synchronized by state correction (changing clocks abruptly) or by rate correction (changing clock speed, but not by too much, so as not to perturb computer operation). In the case of rate correction we need to distinguish between internal and external synchronization: internal synchronization is a cooperative activity among all the nodes of a cluster, whereas external synchronization is a process requiring an external time.

Computer clock synchronization methods have changed over time, directly influenced by technological progress and technical requirements. This section gives a short history of computer clock synchronization leading up to our method. We describe only software synchronization, excluding hardware clock synchronization, since we wish to avoid the necessity for much additional hardware. Our method is intended for fast prototyping, where it is easy to add or remove devices, even during the course of experiments.

2.1.1 Clock model

The first concept to be defined is that of a computer clock. A computer clock is a hardware device subject to physical imperfections. It has been modeled mathematically. A computer clock is composed of an oscillator and a counter. The oscillator generates a periodic process incrementing the counter value. However, the oscillator is not perfect: it does not run at the same speed in every device, and can also run faster or slower within the same device, principally as a result of temperature variations, but also depending on aging and other environmental conditions.

The drift rate (called simply *drift* in this paper) of a clock at time t is defined as the deviation of its speed from the “correct” speed. A perfect clock would have a drift of 0. The drift is defined by

$$\rho_i(t) = \frac{\mathrm{d}h_i(t)}{\mathrm{d}t} - 1 \; . \tag{1}$$

The drift is given at a time t, and can be different at $t' \neq t$. It varies according to different parameters: mainly the temperature, but also external conditions, e.g. vibrations, the aging of the component and other non-monitorable parameters. For example, we observed that when a computer is switched on after a “long period” of inactivity, the clock drift varies significantly over the first 15 minutes or so, and then remains stable at normal room temperature.

A clock's time advances as "correct" time elapses, with its progression modulated by the drift. That is to say, when two clocks i and u are set to the same time at t ($h_u(t) = h_i(t)$), and if u is a perfect clock, the time of clock i at time t' is

$$h_i(t') = h_i(t) + \int_{m=t}^{t'} [\rho_i(m) + 1] \tag{2}$$

Moreover, if between $m = t$ and $m = t'$ $\rho_i(m)$ is constant ($\rho_i(m) = \rho_i$), we have

$$h_i(t') = h_i(t) + (t - t') \times (\rho_i + 1) \tag{3}$$

2.1.2 Computer clock synchronization

Computer clock synchronization with no specification of precision: One of the best-known examples of synchronization is the Network Time Protocol (NTP) [Mills, 1991]. This was originally intended to synchronize computers linked via Internet networks. [Mills, 2003] presents the history of *NTP*. This method uses the *IM* algorithm , explained later. *NTP* is an external synchronization using rate correction: all computer clocks are synchronized to an external reference time, given by the Internet network. The fourth version of the protocol allows a computer clock to be maintained with a precision of 10 ms over the Internet, and 200 μs or less via a local network under ideal conditions.

Some work has been devoted to improving synchronization precision, e.g., *Reference Broadcasts Synchronization* (*RBS*) [Elson et al., 2002]. The idea is that the sender broadcasts a synchronization "pulse" to the other nodes. This pulse is taken as the reference time for all nodes other than the sender.

With the appearance of sensor networks and the timestamping of transmitted messages, the *Timing-sync Protocol for Sensor Networks* (*TPSN*) [Ganeriwal et al., 2003] has become possible: a reference message is sent, and unlike *TPSN*, all nodes change their clock time. Message transmission can therefore be evaluated, leading to better synchronization results.

With these methods, synchronization precision is very good provided that the nodes have been communicating long enough among themselves: we have seen that the synchronization speed with rate correction is very slow. However, the synchronization precision is not explicitly given.

Clock synchronization with precision specification: In the case of dynamic structures (where devices can leave or join the network), reliable communication is harder to attain. Here it is necessary to know the synchronization precision between the different nodes. Synchronization precision can be probabilistic or guaranteed.

Adaptive Clock Synchronization in Sensor Networks (*ACSSN*) [PalChaudhuri et al., 2004] is an *RBS* extension, sending several messages instead of a single one. It is a probabilistic method adapting its precision and power consumption to the application. It gives a good synchronization precision probability and is a good compromise between precision and consumption.

Some methods, such as the *IM* algorithm [Marzullo and Owicki, 1983], place external clock synchronization with interval correspondence in the sensor network domain. Interval correspondence enables the synchronization precision to be given and interval constraint propagation to be used, as explained in [Jaulin et al., 2001]. These methods use state correction, allowing instantaneous synchronization and rapidly decreasing interval imprecision. *NTP* uses the *IM* algorithm, but with rate correction and not state correction, thus avoiding the drawbacks of rate correction resulting from breaks in monotony.

[Kopetz et al., 2006] gives a solution for very precise synchronization (with a granularity of less than one microsecond in multi-cluster systems) for fault-tolerant distributed systems, by combining clock-state and clock-rate corrections. This method is intended for real-time applications lacking very precise expensive clocks, including mass production markets, such as the emerging automotive market for drive-by-wire systems. However, in the field of fast prototyping (the intended scope of this paper), flexibility is required so that one component may quickly be replaced by another, in both experimental and non-experimental settings. In contrast, the real-time domain subject to a large number of constraints which make it ill-adapted to the kind of flexibility which is our concern in this paper.

Time synchronization methods are divided into two families: those which perform rate correction, and those which perform state correction. To obtain optimal precision, in the first case, clocks must be connected over a sufficiently long period. In the second case, the precision is immediately optimal, but there are breaks in clock monotony. Time correspondence was developed to overcome these drawbacks.

2.1.3 Timestamping correspondence

Timestamping correspondence without precision specification: Timestamping correspondence methods have been developed to accompany the increasing popularity of sensor networks, where an exact synchronization is impossible owing to the network dynamic and to the unreliability of communications.

Post facto synchronization [Elson and Estrin, 2001] has been developed to make a clock correspondence between different computers. The method resembles *ACSSN*: in order to timestamp an event with a reference clock a reference time is sent. However, unlike *ACSSN*, no precision is given.

Timestamping correspondence with precision specifications: Some methods provide timestamping correspondence with precision specification, through the use of interval timestamping. These methods were developed for ad-hoc networks, where the network configuration can change, where the propagation time cannot be known in advance, and where consequently it is difficult to make use of a common time (and therefore an external synchronization) because the accessibility of this common time cannot be guaranteed. Timestamping correspondence methods were developed to solve these problems. These methods exchange messages with a view to timestamping particular events.

A first method is presented in [Römer, 2001], which exchanges messages to compare the clocks at the different network nodes and to timestamp events occurring within these nodes. Message exchanges always include a message transmission and an acknowledgement. An event is timestamped using a particular clock, and the timestamp is then converted to another clock.

It will be noticed that the *IM* algorithm (and the *NTP* enhancements) can also be used to convert timestamps from one reference clock to another: *IM* computes an estimation of the clock correspondences with an interval. Instead of synchronizing the clocks, it is possible to use the information to convert a date from one reference clock to another, taking account of imprecision.

2.1.4 General requirements

The requirements are the ability to timestamp events precisely with different clocks and to plug and unplug devices (including devices which may already have their own timestamp conversion system) using a synchronous bus-network.
In other words, we should like to make it possible for two unconnected timestamp-sharing systems to be joined together and share their timestamps with the minimum imprecision added. This requirement makes a timestamp conversion method necessary, because of the dynamic network structure. In order to know the general imprecision, we also need to estimate this imprecision as accurately as possible.
The networks which we are concerned with are synchronous bus-networks, which, by definition, will already posses their own integrated network clocks.

Similar existing systems: A system similar to ours, using the same synchronous bus-network, is described in [Hosek, 2005]. In this work, a special mode of the *Firewire* bus-network is used, where the master interface node (the interface node corresponding to the bus-network clock) may be selected. For this type of bus there is a hi-resolution time ("cycle-time") and a low-resolution time ("bus-time"). Combining these two registers gives a clock with resolution of 40 ns that will overflow every 136 years.
However, in the work just presented, there is a master node and slave nodes: at the outset all nodes must be connected to the master node to have the same time as the master node. Otherwise, if a component were to join the network subsequently (and therefore not have the same time as the master node), there would be a time discontinuity, because of the abrupt clock synchronization. Moreover, during a disconnection period, the created subnetwork clocks will drift apart from each other.

In our work we want it to be possible for two networks to join together even if they have not previously been connected. When disconnections and reconnections occur, we wish to estimate the error due to clock drift between the two networks. Moreover, we wish to estimate the imprecision due to timestamp conversion. These requirements are not met by the previously-cited work.

2.2 Basis of our approach

The system is intended for the fast prototyping of applications with numerous sensors and a large data flow, requiring a precise timestamping and a distributed architecture. Interval timestamping is used here to estimate timestamp imprecision. This can be useful for integrity checking. By "fast prototyping" we mean the experimental deployment of applications with low hardware costs, enabling cooperative computers to be used and different configurations to be tested without the need for large, expensive systems which would be difficult to implement. A distributed architecture can also be used to improve the real-time behavior of the system: the greater the computer workload, the slower its reactivity.

In this section we present the two main characteristics of the proposed system: interval timestamping and a synchronous network. The system is the same as that presented in [Bezet and Cherfaoui, 2005a] and [Bezet and Cherfaoui, 2005b].

2.2.1 Timestamping with intervals

The paradigm of interval-based synchronization in the context of infrastructure-based networks was proposed by Marzullo and Owicki [Marzullo and Owicki, 1983] and refined by Schmid and Schossmaier [Schmid and Schossmaier, 1997]. This paradigm was introduced to evaluate the uncertainty caused by the time reference change.

The concept of interval date is extended in this paper to timestamps. It provides guaranteed bounds between which a particular event is certain to have occurred. The difference between the upper and the lower bound (interval width) is called the uncertainty.

Interval dates help to overcome several imperfections. The first imperfection is the granularity of the computer clock: a given date means nothing more than somewhere in the interval between two ticks of the system clock. The second imperfection is the inherent limitation of hardware components, for example latency and transmission lags of sensors and computers, which means data cannot be dated accurately.

Algorithms are nowadays more and more accurate and efficient. If there are no good timestamping estimations, it becomes futile to continue improving the accuracy of algorithms, since results can never be satisfactory.

Interval timestamping is also useful for checking data integrity. Data integrity verification can be important, e.g., for data fusion processes. It enables to system coherence to be controlled: if there is a problem of integrity, data can be wrong. A date approximation can cause an integrity problem, whereas interval timestamping means that dating is certain (an example of a timestamp integrity problem is given for the case of a laser range scanner in [Bezet and Cherfaoui, 2006]).

Furthermore, the combination of several bounds for a single time is unambiguous and optimal, whereas the combination of time estimates requires additional information about the quality of the estimates, if it is to be unambiguous.

In spite of its advantages, timestamping with guaranteed bounds has not been extensively studied.

Interval analysis: As intervals are used, we shall now provide some explanations regarding interval analysis. This section is freely inspired from the book [Jaulin et al., 2001], which provides the basis for the interval analysis used in this paper.

An interval is denoted $[x]$ or $[\underline{x}, \overline{x}]$, meaning that $[x]$ is a number between $\underline{x}$ and $\overline{x}$. The difference between the upper bound $\overline{x}$ and the lower bound $\underline{x}$ is called the interval width $= \overline{x} - \underline{x}$. Given two intervals $[x]$ and $[y]$, the basic arithmetic operations, addition, subtraction, multiplication and division used in this paper are defined as

$$[x] + [y] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \tag{4}$$

$$[x] - [y] = [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \tag{5}$$

$$\begin{aligned}[x] \times [y] = \; & [\min(\underline{x} \times \underline{y}, \underline{x} \times \overline{y}, \overline{x} \times \underline{y}, \overline{x} \times \overline{y}), \\ & \max(\underline{x} \times \underline{y}, \underline{x} \times \overline{y}, \overline{x} \times \underline{y}, \overline{x} \times \overline{y})]\end{aligned} \tag{6}$$

$$1/[x] = \begin{cases} [1/\overline{x},\ 1/\underline{x}], & \text{if } 0 \notin [x] \quad (7) \\ [-\infty, +\infty], & \text{if } 0 \in [x] \quad (8) \end{cases}$$

$$[x]/[y] = [x] \times 1/[y] \tag{9}$$

Interval analysis enables constraint propagation. It enables the width of an interval to be reduced by taking advantage of redundant data intervals. It uses the property of an interval number which ensures that the value is between the lower and the upper bound of the interval. If several interval numbers with the same value are computed, the intersection of the intervals gives another interval with a reduced interval width.

Note: all calculations in this paper are done using intervals.

2.2.2 The synchronous network

Synchronous networks were developed to avoid message collisions, increasing the data flow [Obermaisser, 2004]. A node cannot send a message at an arbitrary time, but must await its turn: this is known as time-triggered communication. A time-triggered communication system uses the Time Division Multiple Access (TDMA) media access strategy. TDMA statically divides the channel capacity into a number of slots and assigns a unique slot to every node. The communication activities of every node are controlled by a time-triggered communication schedule. The schedule specifies the temporal pattern of message transmissions, i.e., at what time point nodes send and receive messages. A sequence of sending slots is called a TDMA round. This enables every member of a set of nodes to transmit a message exactly once. The implementation of such a method requires a global clock. As far as we know, no timestamp conversion has ever been developed for a synchronous network; that is what motivated this paper.

Being able to plug and unplug hardware is desirable when fast prototyping is used: this means that new sensors, computers and subnetworks may be added to the existing network during experiments, and dysfunctional switches may be taken into account.

3 Timestamp conversion with bounded error specification

The proposed methods for on-line and post-processing timestamp conversion are based on the same principles:

- All data are timestamped using the local system clock.
- All timestamps are intervals: they give guaranteed bounds within which it is certain that events occurred.
- At each data exchange, a new timestamp interval is computed with the new local clock.
- The new timestamp takes into account the drift between the different clocks in order to give a better estimation and to guarantee the interval estimation.
- Computations are performed using interval analysis, providing guaranteed bounds on the timestamps. Timestamp intervals reflect the time correspondence precision.

Note: the network communication mode is client-server architecture, where clients request data provided by servers. Servers produce the original data timestamps, and clients receiving the data must compute new timestamps in accordance with their local clock reference.

On-line timestamp conversion and post-processing timestamp conversion use the same system architecture (various computers are connected via a synchronous bus). The client-server architecture is used on-line to transfer time correspondence information when timestamp conversions are required. For post-processing conversion, client-server architecture is not essential, and the timestamp conversion information is stored via time correspondence computations based on readings of synchronous network time. The main problem here is managing possible insertions/removals of new subnetworks with different clock times. The two conversions use the same basic principles, and can be managed simultaneously.

3.1 Proposed approach for on-line timestamp conversion

The general principle of on-line timestamp conversion is as flows: the server converts the data timestamps from its local system time base to the synchronous network time base. They are then sent to the client, which converts them from the synchronous network time base to its local system time base.
This process is completely extensible: one computer can have several servers and/or clients. Likewise, the server and its client(s) can be implemented on the same computer.

3.1.1 Computation of the new timestamp

The process is divided into two steps. The first step, performed by the server, is to convert the timestamp from the server clock's reference time to the synchronous network clock reference time. The second, performed by the client, is to convert the received time to the client clock's reference time.

If we consider time periods during which conditions are sufficiently similar for us to assume constant drift in the involved clocks (i.e. where the temperature does not vary significantly and without significant vibrations), Equ. (3) can be used to convert the time from one clock reference to another. For time t_a, t_b, with $t_a < t_b$, this drift is estimated with the average drift $\hat{\rho}_{i/j}$ in $[t_a,\ t_b]$ $\big($denoted $\hat{\rho}_{i/j}(t_a, t_b)\big)$ as

$$\hat{\rho}_{i/j}(t_a, t_b) = \frac{h_i(h_j(t_b)) - h_i(h_j(t_a))}{h_j(t_b) - h_j(t_a)} - 1 \ . \tag{10}$$

As $h_i(h_j(t))$ is $h_i(t)$ (t is an absolute time), the previous equation can be written

$$\hat{\rho}_{i/j}(t_a, t_b) = \frac{h_i(t_b) - h_i(t_a)}{h_j(t_b) - h_j(t_a)} - 1 \ . \tag{11}$$

$\hat{\rho}_{j/i}(t_a, t_b)$ can easily be computed from Equ. (11) as

$$\hat{\rho}_{j/i}(t_a, t_b) = \frac{1}{1 + \hat{\rho}_{i/j}(t_a, t_b)} - 1 \ . \tag{12}$$

However, exact times are unknown, which is why we use interval times instead. The time correspondence between two clocks i and j can be approximated by taking the time of a clock i, then the time of the other clock j and finally the time of the first clock i, giving an interval. This process prevents a possible interruption of unknown duration between the two clock readings. For Equ. (11) we obtain a drift estimation of

$$[\hat{\rho}_{i/j}(t_a, t_b)] = \frac{[h_i(t_b)] - [h_i(t_a)]}{[h_j(t_b)] - [h_j(t_a)]} - 1 \ . \tag{13}$$

Moreover, the correspondence between two computer clocks is unknown. This is why two steps are needed: first the conversion from the server to the network clock, and then from the network clock to the client clock.

To convert a timestamp from the server to the client clock reference, three clocks are needed: the server clock, the client clock and the synchronous network clock. Their respective readings at time t are denoted as $h_s(t)$, $h_c(t)$ and $h_n(t)$ (notation of [Blum et al., 2004]. The local clock (server or client clock) is denoted as $h_l(t)$. It should be noted that t_a, t_b and t_c are not the same times for the client and for the server. They are denoted t_{a_c}, t_{b_c}, t_{c_c} and t_{a_s}, t_{b_s}, t_{c_s} for respectively the client and the server. However, t is the same for the client and the server.

The first step is to convert the timestamp from the server reference time to the synchronous network reference time. The corresponding equation is

$$[h_n(t)] = [h_n(t_{c_s})] + ([h_s(t)] - [h_s(t_{c_s})]) \times \left[1 + [\hat{\rho}_{n/s}(t_{a_s}, t_{b_s})]\right] \tag{14}$$

The second step is to convert the timestamp from the synchronous network reference date to the client reference date. The corresponding equation is the symmetric of Equ. (14), i.e.

$$[h_c(t)] = [h_c(t_{c_c})] + ([h_n(t)] - [h_n(t_{c_c})]) \times \left[1 + [\hat{\rho}_{c/n}(t_{a_c}, t_{b_c})]\right] \tag{15}$$

3.2 Proposed approach for post-processing timestamp conversion

The general idea behind post-processing timestamp conversion is to regularly record clock equivalencies during experiments, enabling timestamps to be interpolated or extrapolated at the post-processing phase from one time base to another, thanks to the recorded files.

3.2.1 Required process during data acquisition and storage

A low priority process must be run during data acquisition, which will then allow the time equivalence between the clocks to be computed during post-processing. This process runs on all computers, and generates a file for each computer, called the *time equivalence file*. This file contains the time equivalencies between the local computer clock and the synchronous network clock. This *time equivalence file* can be written for example at the frequency of 1 Hz (this frequency depends on the network dynamic). As time equivalencies cannot be known with an exact manner, interval dates are written into the file. These interval dates are used to compute the clock correspondences.

As the synchronous network clock is not monotonous, a monotonous time is required so that computer clock equivalencies may be computed without any ambiguity. This time is the time of one of the computers (C_r) composing the network. It is the same for all computers linked to the network, and the particular computer to be used may be chosen arbitrarily. This allows possible ambiguities and defective links to be revealed. The insertion of the reference computer in the *time equivalence file* requires a relatively small amount of network bandwidth. If margins are so tight that it is not possible to obtain any additional network bandwidth, this field may be dropped. It is only a piece of information used to simplify post-processing calculations.

The *time equivalence file* is composed of several records. Each record comprises five fields, as shown in Tab. 1: the local computer time is first taken (time t_{l1}), then the *net-time* (time t_n), next the local computer time again (time t_{l2}, and $\overline{t_{l2}} - \underline{t_{l1}}$ is stored, to save space). Finally, a request is transmitted in order to obtain the reference computer name (C_r) and time (t_{C_r}, which is an approximation of the reference computer time expressed as a point time).

Local time	Net time	Diff local time	Reference computer name	Reference computer time
$\underline{t_{l1}}$	t_n	$\overline{t_{l2}} - \underline{t_{l1}}$	t_{C_r}	C_r

Table 1: A *time equivalence file* record

As the *net-time* (synchronous network clock) is incremented by a software counter, this counter needs to be the same for all nodes linked to the network. This constraint is satisfied by resetting the software counter at every addition or removal of a component, when the network time is synchronized abruptly. When the network architecture changes, a bus-reset is sent: when a bus-reset is detected, the *net-time* software counter is set to zero, which means that all nodes connected to the network always have the same network time.
At each bus-reset detection, a special record (*bus-reset record*) is written to the *time equivalence file*, so that between two bus-reset records we have a block where the *net-time* is monotonous, and comes from the same free-running clock.

3.2.2 Time correspondence computation

The goal is to obtain the time equivalence between two local computer times. This computation makes use of two *time equivalence files.* Checking that two computers have been linked simply means checking that the same reference computer is to be found in the two files, with a non-null reference time interval intersection. Once two such records have been detected in the files, it is certain that the two related blocks correspond to an identical sequence.

If no reference computer is to be found in the *time equivalence file*, finding two equivalent blocks involves a little more effort. We have to check two equivalent blocks where the *net-time* begins and ends at approximately the same value, which is not a completely sure method. However, the probability of performing two experiments where the *net-time* begins and ends at the same values is very low. If, moreover, there are bus-resets in the file (i.e. architecture changes), the probability of having the same values for two files where the nodes have not been connected is close to zero. However, to reduce this error probability, the write-frequency of the *time equivalence file* can be increased, or random bus-resets can be generated.

3.2.3 Time equivalence computation between two clocks

Before we describe time equivalence between two clocks, it might be useful to present a first figure illustrating the interval notation used in subsequent figures. Fig. 1 illustrates clock time estimation with intervals. The X-axis represents a *reference time*, the reference for all other clock times. This time cannot be known exactly, but it exists. Clock A's time progression is represented by the plain line. If the drift is considered constant, the clock time progression is an affine function. Then, at times t_1 and t_2 clock A's time is estimated using interval times, giving respectively $[t'_1]$ and $[t'_2]$. Between t_1 and t_2, clock A's time progression can be estimated with an upper and lower line, with an interpolation between t_1 and t_2, and an extrapolation outside (dashed lines in the figure).

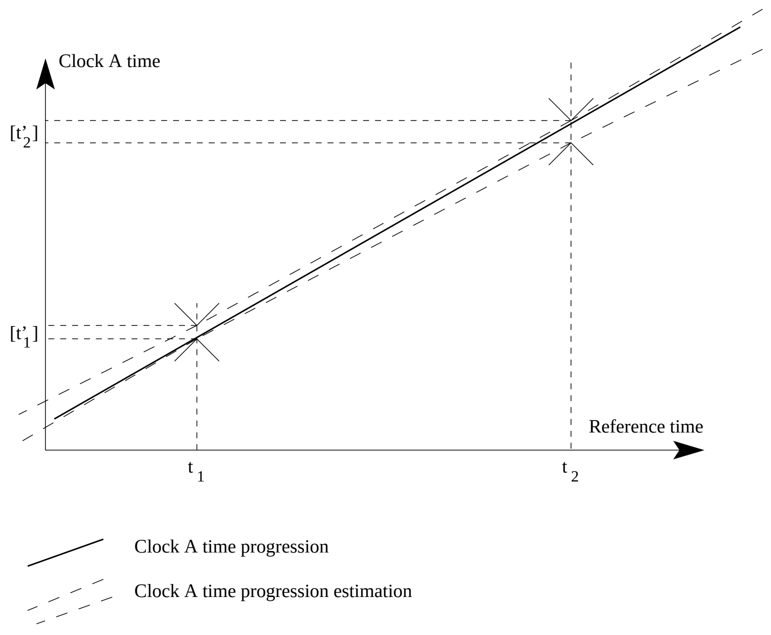


Figure 1: Clock time estimation

When a time is estimated for a given clock,

Fig. 2 shows time equivalencies recorded in the *time equivalence file* between clocks A, B and the network clock. Interval times are represented and a possible progression line is represented for each clock. The objective is to compute $[T_{A_{NB_2}}]$ for clock A when clock B has the value $[T_{B_2}]$, using the network clock $[T_{NB_2}]$ at time $[T_{B_2}]$ as seen by clock B.

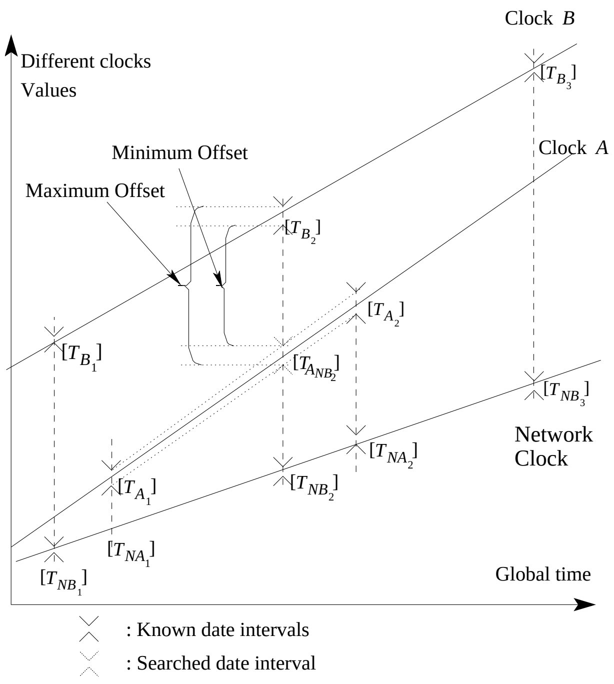


Figure 2: Timestamp correspondence for two *time equivalence files*

To compute $[T_{A_{NB_2}}]$ the drift is assumed to be continuous and constant between $[T_{A_1}]$ and $[T_{A_2}]$ (there must be no network time discontinuity between $\underline{T_{NA_1}}$ and $\overline{T_{NA_2}}$). The time correspondence in the network is known, $[T_{NB_2}]$. The only computation necessary is the conversion of $[T_{NB_2}]$ to $[T_{A_{NB_2}}]$. If the network time is not continuous between $[T_{A_1}]$ and $[T_{A_2}]$ (as in Fig. 3), this conversion is impossible.

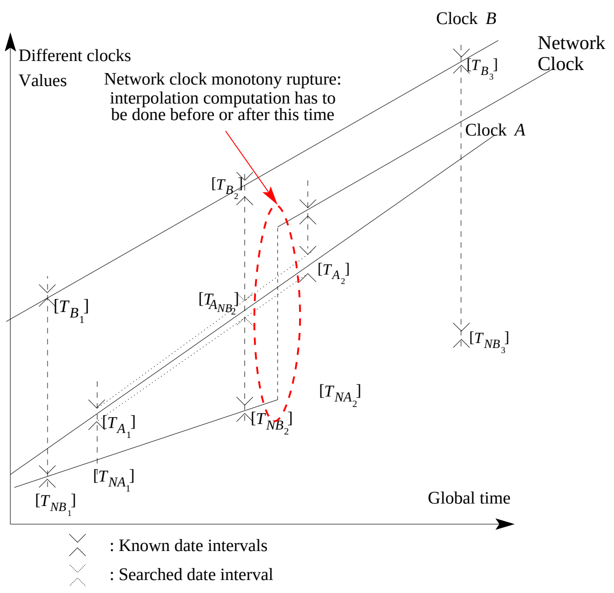


Figure 3: Bad timestamp correspondence computation for two *time equivalence files*

In Fig. 3, network clock monotony is interrupted between $[T_{A_1}]$ and $[T_{A_2}]$, following the insertion of another network: the two networks set their clocks to the same time, producing a break in monotony. The computation has to be performed before or after the point of interruption.

In Fig. 4, minimum and maximum bound segments for the correspondence between the network and clock A times are computed. We can see that in the network clock time segment under consideration, there are two possible time segments for clock A. This results from a break in network time monotony, possibly following the addition or removal of a network interface. The problem can be resolved through careful handling of time monotony.

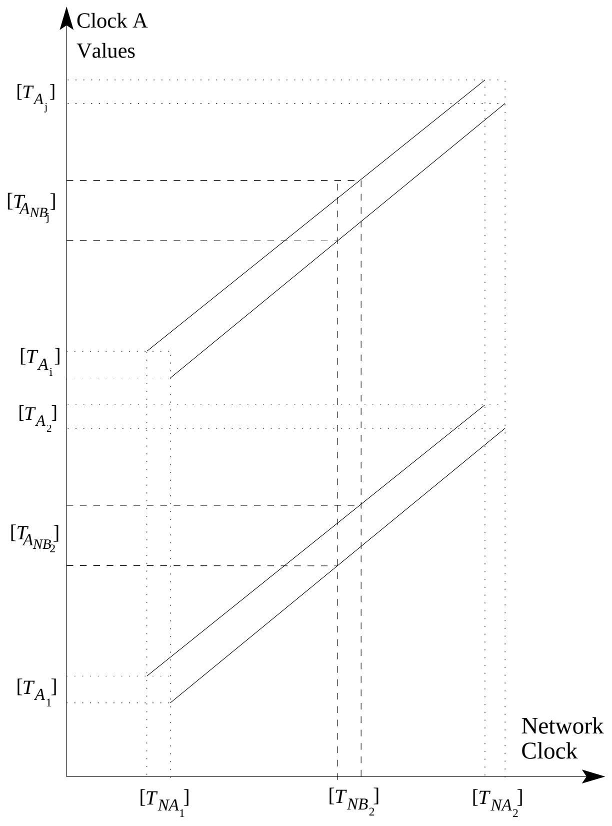


Figure 4: Timestamp correspondence from the network time to clock A time, for two *time equivalence files*, over a time interval where the network clock gives two results for clock A time

We now consider the time segment giving $[T_{A_1}]$ and $[T_{A_2}]$. $[T_{A_{NB_2}}]$ can be computed if the network clock is continuous between $\underline{T_{NA_1}}$ and $\overline{T_{NA_2}}$, and similarly for the lower bound

$$\underline{T_{A_{NB_2}}} = \underline{T_{A_1}} + (\underline{T_{A_2}} - \underline{T_{A_1}}) \times \frac{\underline{T_{NB_2}} - \overline{T_{NA_1}}}{\overline{T_{NA_2}} - \overline{T_{NA_1}}}, T_{NB_2} \in [\underline{T_{NA_1}}, \overline{T_{NA_2}}] \tag{16}$$

and for the upper bound

$$\overline{T_{A_{NB_2}}} = \overline{T_{A_1}} + (\overline{T_{A_2}} - \overline{T_{A_1}}) \times \frac{\overline{T_{NB_2}} - \underline{T_{NA_1}}}{\underline{T_{NA_2}} - \underline{T_{NA_1}}}, T_{NB_2} \in [\overline{T_{NA_1}}, \underline{T_{NA_2}}] \tag{17}$$

3.2.4 Computation of the clock correspondence function

A correspondence function can be computed using equivalence points. These enable time equivalencies to be computed rapidly with the aid of an interpolation function. A sufficient number of time equivalencies between clocks A and B need to be computed over time intervals where the drift is constant. Fig. 5 is an example of time equivalence between clocks B and A, where the drift is constant between $[T_{B_1}]$ and $[T_{B_2}]$. A time $[T_{B_\alpha}]$ must be converted from clock B to A reference time, giving $[T_{A_{T_{B_\alpha}}}]$. This is quite similar to Fig. 4, except that the function computation is between clock B and A, and not between the network clock and clock A. There cannot therefore be several time correspondences between clock A and clock B times: these two clocks are monotonous.

For the lower bound (see Fig. 5) the computations are:

$$\underline{T_{A_{T_{B_\alpha}}}} = \underline{T_{A_1}} + (\underline{T_{A_2}} - \underline{T_{A_1}}) \times \frac{\underline{T_{B_\alpha}} - \overline{T_{B_1}}}{\overline{T_{B_2}} - \overline{T_{B_1}}} \tag{18}$$

and for the upper bound:

$$\overline{T_{A_{T_{B_\alpha}}}} = \overline{T_{A_1}} + (\overline{T_{A_2}} - \overline{T_{A_1}}) \times \frac{\overline{T_{B_\alpha}} - \underline{T_{B_1}}}{\underline{T_{B_2}} - \underline{T_{B_1}}} \tag{19}$$

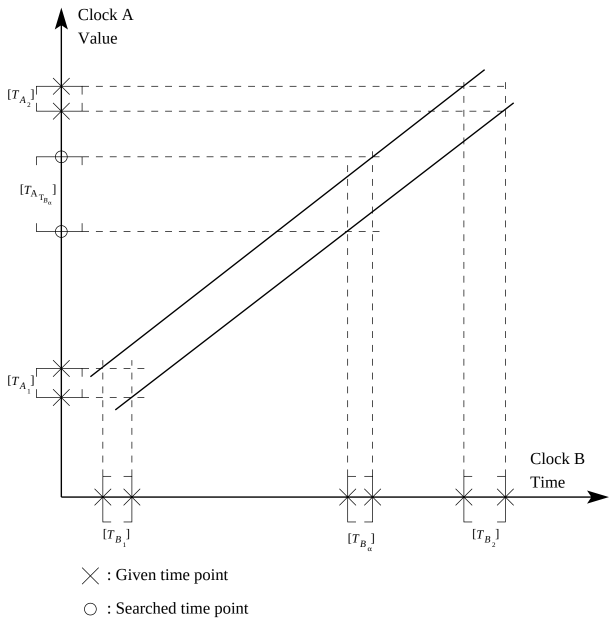


Figure 5: Equivalence function between clock B and clock A times

The sequence to convert is divided into intervals where the drift is constant: if the drift is constant over the whole sequence, two equivalence points are enough. Otherwise, it is divided into smaller intervals. The drift linearity hypothesis can be checked by analyzing the *time equivalence files*: the interval drift is estimated between all data records, and if the interval drift intersection is null, we may conclude that the drift estimations are not the same at the different times. The sequence therefore needs to be linearized in order to yield a constant drift between the equivalence points.

If time equivalencies do not occur both before and after the time to be converted, an extrapolation is required, as shown in Fig. 6. In this figure, the time $[T_{B_\beta}]$ to be converted is before the first equivalence point $\{[T_{A_1}],[T_{B_1}]\}$, so the computation is performed using the closest equivalence point $\{[T_{A_1}],[T_{B_1}]\}$, to which is added to the time difference to this point, and the result is adjusted according to the drift estimation. In the figure the drift estimation is represented by the upper and lower straight lines to the left of the point $\{[T_{A_1}],[T_{B_1}]\}$.

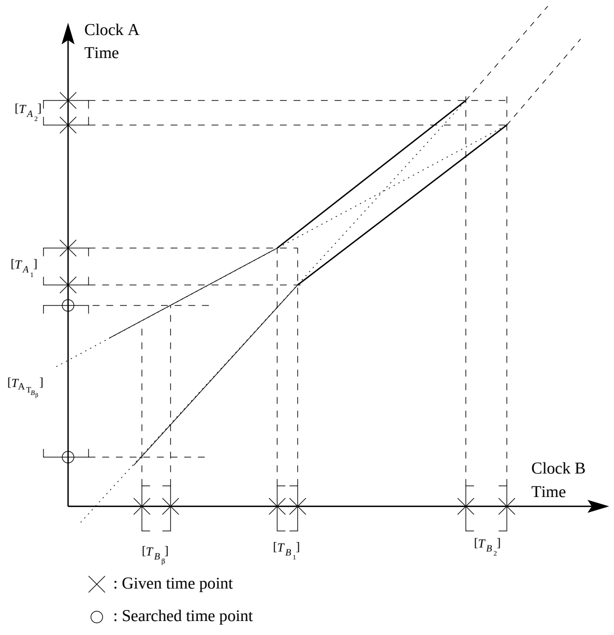


Figure 6: Example of extrapolation before the first found equivalent point

In Fig. 6, the first time equivalence found between the two clocks A and B is the couple $\{[T_{A_1}],[T_{B_1}]\}$. T_{B_β} does not have equivalence points occurring both before and after, but only one equivalence point after, and so an extrapolation is required; the method is similar to the on-line time conversion described in Sect. 3.1.1, except that one only step is required: the direct conversion from clock B to clock A, because some equivalencies of clock B are known in relation to clock A's reference time.

The method works if the drift is more or less linear between the various times, as we have shown above, and involves computing the equivalence time with the closest time equivalence, adding the time difference with respect to the time to convert, and correcting using the drift. The drift is evaluated with two time equivalencies between the clocks, where the drift is constant. In Fig. 6, these equivalent points are $\{[T_{B_1}], [T_{A_1}]\}$ and $\{[T_{B_2}], [T_{A_2}]\}$. So the drift is computed as:

$$[\widehat{drift_B}\_{}_A(T_1, T_2)] = \frac{([T_{B_2}] - [T_{A_2}]) - ([T_{B_1}] - [T_{A_1}])}{[T_{B_2}] - [T_{B_1}]} - 1 \tag{20}$$

The computation is almost the same for timestamps before the first found equivalence time $\{[T_{A_1}], [T_{B_1}]\}$ as for timestamps after the last found equivalence time $\{[T_{A_n}], [T_{B_n}]\}$, except that $[T_{A_1}]$ and $[T_{B_1}]$ are respectively replaced by $[T_{A_n}]$ and $[T_{B_n}]$. So the computation gives

$$[T_{A_\beta}] = [T_{B_\beta}] + [T_{A_1}] - [T_{B_1}] + ([T_{B_\beta}] - [T_{B_1}]) \times [\widehat{drift_B}\_{}_A(T_1, T_2)] \tag{21}$$

Since interval notation is used, interval propagation may be used to improve the drift estimation. As the drift is estimated in a particular manner and can be estimated at different points (leading to different interval results), the intersection between the resulting intervals can be performed, giving a better drift estimation interval. A null intersection implies that the drift is not linear over the period under consideration: this period used for drift estimation must be lengthened, in order to linearize the drift.

4 Theoretical and experimental performances

Simulations and experiments were carried out to estimate the performances of the proposed methods. Computations estimate the resulting interval timestamp width. This width depends on the various parameters influencing the equations computing the resulting intervals explained in section 3.

To simulate and experiment the performances, a clock granularity of 1 μs is assumed.

4.1 The bus-network used

For our experiments we chose a *FireWire* bus network in preference to others (e.g., CAN, TTP, Flexray), for several reasons. First, the application was not designed to be embedded in mass production cars, but to be used in the development of data acquisition and data computation for new *ADAS* (Advanced Driver Assistance Systems) functions, or for studying driver behavior, i.e., fast prototyping. Secondly, the network requires a large bandwidth to be able to transmit video streams. Finally, the *FireWire* bus network is able to process dynamic reconfigurations.

The *FireWire* bandwidth is 400 Mb/s for IEEE 1394a [Anderson, 1999]. It was originally designed for video data transfer. It has two data transfer types, asynchronous and isochronous. The *FireWire* has a synchronous bus network clock with a frequency of 24.576 MHz for IEEE 1394a. Each interface has a clock synchronized with a precision of less than 5 μs for all interfaces, depending on the overall cable length.

The *FireWire* global clock is the clock of one of the interfaces forming the *FireWire* bus network, and consequently a free-running clock. All interfaces are synchronized to this reference clock, whose frequency is not modified. The synchronous network clock is used by the computers to bring about the desired time correspondence.

In spite of all its advantages, the *FireWire* clock (like all synchronous network clocks) is not monotonous for two reasons. First, the *FireWire* clock has a counter capacity of approximately 128 s, and secondly, there is a break in monotony at each bus reset, which can be a consequence of the plugging or unplugging of an interface.

To prevent overflows we added a software counter to the interface counter. The synchronous network clock with the addition a software counter is called the *net-time* in this paper. This software counter is a 32-bit integer, which allows more than 17000 years to elapse before a buffer overflow occurs.

A second type of break in monotony may be caused by the plugging or unplugging of hardware. These events lead to a new network configuration: the *FireWire* clock can change (the *FireWire* clock is in fact one of the interface clocks). This possibility must be taken into account when converting timestamps.

4.2 Theoretical performances

4.2.1 Theoretical performances of on-line timestamp conversion

On-line timestamp conversion depends on the parameters of Equ. (14) and (15).

Fig. 7 shows the influence of the various parameters on on-line interval width estimation. The graph is plotted with the parameters of Tab. 2, with ${}^{w}[t] = \bar{t} - \underline{t}$ the interval width of $[t]$. To simplify the resulting interval width estimation, it is assumed that varying parameters have values between 5 μs and 500 μs.

Curve	Value	Default value
(1)	${}^{w}[\hat{\rho}_{n/s}(t_{a_s}, t_{b_s})] = {}^{w}[\hat{\rho}_{c/n}(t_{a_c}, t_{b_c})]$	1 μs
(2)	${}^{w}[h_n(t_{c_s})] = {}^{w}[h_n(t_{c_c})]$	6 μs
(3)	${}^{w}[h_s(t)]$	3 μs
(4)	$t - t_{c_c} = t - t_{c_s}$	2.10^6 $\mu s = 2s$

Table 2: Values used for the graphs in Fig. 7
(1): Drift interval width
(2): Network interval width
(3): Original timestamp interval width
(4): Time delay between the timestamp and the data transfer

The value of the drift is also influent: this modifies the last multiplication parameter in Equ. (14) and (15). However, for values varying from -200 $\mu s/s$ to +200 $\mu s/s$ (maximum observed drifts), the theoretical influence of the drift on the resulting timestamp is much lower than 1 μs, which is negligible in relation to the other influent parameters.

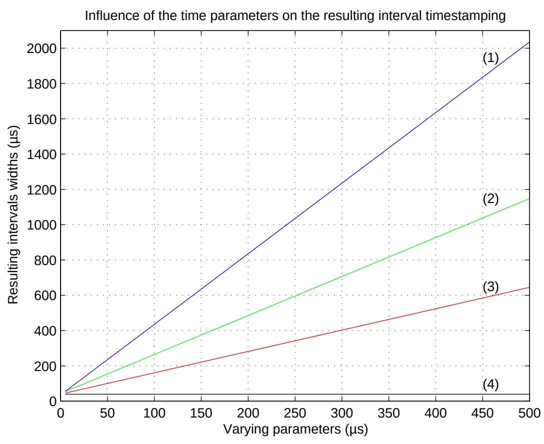


Figure 7: Influences of the various parameters on the resulting timestamping interval width

The resulting timestamping interval width has an optimum value of about 45 μs, and the resulting interval width varies linearly with respect to the various parameters.

It will be noticed that the least influent parameter is the delay between the timestamp and the data transfer, because this parameter influences only the size of the multiplication factor at the end of the equations. It is not very influent.
Neither is the original timestamping interval width particularly influent, because this value is taken into account only once (at the first computation), and so the error is not cumulative through the different equations.
The network interval width is approximately twice as influent as the original timestamping interval width, because this error is taken into account twice in the equations, for the computation from the server time to the network time, and from the network time to the client time.
Finally, the most influent error is the drift interval width of the different clocks, approximately four times more influent than the original timestamping interval width.

Fig. 8 shows the influence of the delay between the timestamp and the data transfer from 0 to 500 s.

4.2.2 Theoretical performances of post-processing timestamp conversion

Theoretical experiments were also carried out to estimate the loss of precision in post-processing timestamp conversion. Post-processing timestamp conversion can be separated into two different methods, depending on the location of the timestamp to convert, implying interpolation or extrapolation. The first step is the same, see Equ. (16) and (17), and then computations are different for the two methods, see Equ. (18), (19) for interpolation and (21) for extrapolation.

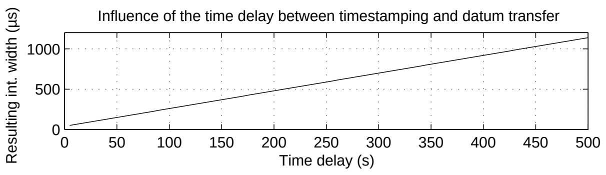


Figure 8: Influence of the time delay between the timestamp and the data transfer on the resulting timestamping interval width

As mentioned in the previous section, some simplifications are made when determining the influence of the various parameters. The main simplification is that timestamping interval widths are assumed to be identical for the server and the client computers at any acquisition time. The same assumption is made for network timestamping widths. This simplification implies that the two corresponding timestamp points at the beginning and at the end of the recorded sequences have the same interval precision width.

We now distinguish between the parameters influencing post-processing interpolation and those influencing post-processing extrapolation.

Theoretical performances of interpolation post-processing conversion: Various influent parameters will be remarked in Equ. (18) and (19). Fig. 9 illustrates the graphs of interval width, depending on the parameters of Tab 3. The parameters vary from 5 μs to 500 μs.

Line	Value	Default value
(1)	${}^{w}[T_{B_\alpha}]$	3 μs
(2)	${}^{w}[T_{A_1}] ={}^{w} [T_{A_2}]$ $={}^{w} [T_{B_1}] ={}^{w} [T_{B_2}]$	3 μs
(3)	${}^{w}[T_{NB_1}] ={}^{w} [T_{NB_2}]$	8 μs

Table 3: Values used for the graphs in Fig. 9
(1): Original timestamp interval width
(2): Local clocks interval width
(3): Network interval width

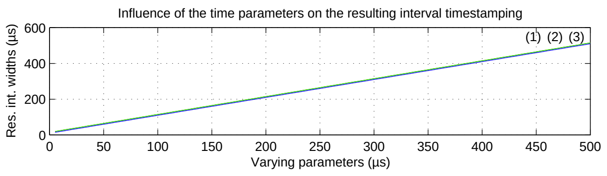


Figure 9: Influence of the various influent interval widths on the resulting interpolation post-processing timestamp interval width

In Fig. 9 the three lines describe almost identical trajectories, with a resulting interval width of about 15 μs for the initial chosen parameters. This interval width varies linearly from 15 μs to about 520 μs for interval widths of 500 μs. The result is almost identical for the three varying parameters, i.e., the resulting interval width depends directly on the various interval widths of the formulae.

The interval width of the interpolation part of the post-processing conversion does not depend on any delays with respect to equivalence points, because the drift is considered as linear during this stage, leading to lower and upper interpolation bounds. That is to say, when the drift has been verified as constant over the period in question, an interpolation is sufficient to convert timestamps from one reference clock to another.

Theoretical performances of extrapolation post-processing conversion: We also provide some graphs corresponding to the extrapolation part of the post-processing conversion. The influent parameters are the same as for the interpolation part, plus the drift and the delay between the closest correspondence point and the time to convert (see Equ. (21)), like in the case of on-line timestamp conversion, Sect. 4.2.1. Values vary from 5 μs to 500 μs and are explained in Tab. 4 for Fig 10.

In Fig. 10, the most influent parameter is the drift interval width -line (1)-. It is responsible for approximately half the interval width error on the resulting timestamping interval width, when compared to Fig. 7. The influence is smaller than in the case of on-line conversion, because in post-processing conversion the drift is estimated directly between the two local computer clocks, whereas in on-line conversion the drift is estimated

Line	Value	Default value
(1)	$^w[\widehat{drift_{B\_A}}(T_1,T_2)]$	$1\ \mu s([100,101]\ \mu s/s)$
(2)	$^w[T_{A_1}] =^w [T_{A_2}]$ $=^w [T_{B_1}] =^w [T_{B_2}]$	$3\ \mu s$
(3)	$^w[T_{NB_1}] =^w [T_{NB_2}]$	$8\ \mu s$
(4)	$^w[T_{B_\beta}]$	$3\ \mu s$
(5)	$^w([T_{B_\beta}] - [T_{B_1}])$	$2\ s$

Table 4: Values used for the graphs in Fig. 10
(1): Drift interval width
(2): Local clocks interval width
(3): Network interval width
(4): Original timestamp interval width
(5): Time delay

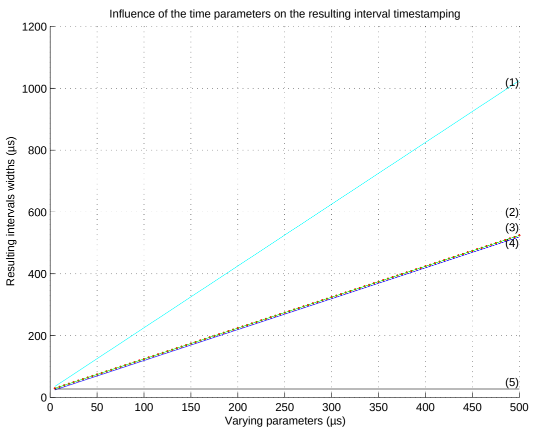


Figure 10: Influence of the various interval widths on the resulting extrapolation post-processing timestamp interval width

between the server clock and the network clock, and then between the network clock and the client clock, implying twice the drift error. For larger values we provide details in Fig. 11.
The middle lines (2), (3) and (4) have approximately the same influence as in Fig. 9, because of the similar situation and because the influence of the drift interval width -line (1)- and the time delay -line (5)- is practically non-existent.
Like in previous graphs, the delay between the reference time correspondence and the time to convert -line (5)- is almost imperceptible for such short delays.

Fig. 11 illustrates the influence of the time delay between the closest correspondence points of the two clocks which are to be set to the same time and the timestamp to be converted. The time delay varies from 5 to 500 seconds.

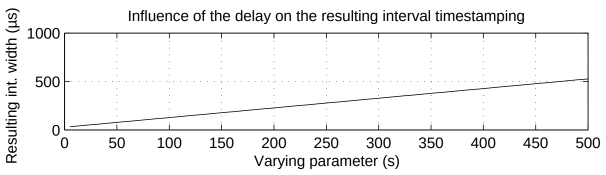


Figure 11: Influence of the various interval widths on the resulting extrapolation post-processing timestamping interval width

In Fig. 11, it will be noticed that the interval width is approximately half that of the implied delay for on-line timestamping, as shown in Fig. 8. The explanation is the same as for drift interval width.

In conclusion, every parameter is important if the best possible resulting timestamping interval widths are to be obtained. The better the initial precision, the better the resulting precision. Furthermore, the drift between the different clocks is twice as significant as regards the interval width for on-line timestamping (as opposed to for post-processing timestamp conversion), given that in post-processing conversion the various parameters can be analyzed to be optimized, whereas in on-line conversion, conversions cannot be analyzed.

4.3 In-lab experimental results

4.3.1 In-lab experimental results for on-line timestamp synchronization

We now present some on-line timestamp conversion experiments performed in the lab.

The system presented here, involving wired networks linked via synchronous bus-networks, and where independent subnetworks are free to join or leave, is in fact unique. This solution is totally distributed, meaning that there is no need for master and slave nodes, and there exist no orders regarding the addition of nodes or networks. Experiments were carried out to show the influence and the consequence of the drift on the resulting timestamping. These experiments showed that the resulting timestamping interval encompassed the original interval. To perform these experiments, two computers were used, with the following scenario. The first computer runs a client program which sends a request to the second computer running a server program. The latter returns a timestamp to the client. The first computer converts the timestamp to its local clock base, as explained in this paper. The timestamp is fixed and does not change: this enables the increase in timestamping interval width to be visible as a timestamp ages.

The synchronous bus network used is *FireWire*, whose clock granularity is set to 5 μs. The global timestamping granularity is set to 1 μs. The data exchange and drift estimation is performed at a frequency of 3 Hz, and when the drift estimation is not performed (e.g. just after a bus reset), or when it is worse than $[-120,\ 120]$ $\mu s/s$, this interval is recorded. Some plugging and unplugging events are carried out to simulate the connection and disconnection of synchronous bus network components. The operating system used for this experiment was *Linux RTAI-LXRT* (*Linux Real Time*).

The protocol used was *SCOOT-R*, a middleware developed in the laboratory [Chaaban et al., 2003][1]. *SCOOT-R* stands for Server and Client Object Oriented for Real Time. It is a system designed for simple object exchanges, using either a client-server or a transmitter-receiver model, both in real time. This software can be used either with *Microsoft Windows* (which is not hard real time) or *Linux RTAI-LXRT* (real time). It also includes a function to generate the previously cited extended bus network time. In-house software could also be used in place of *SCOOT-R*.

The experiment described was carried out over more than 6 minutes. The new computed timestamps were recorded and are shown in Fig. 12.

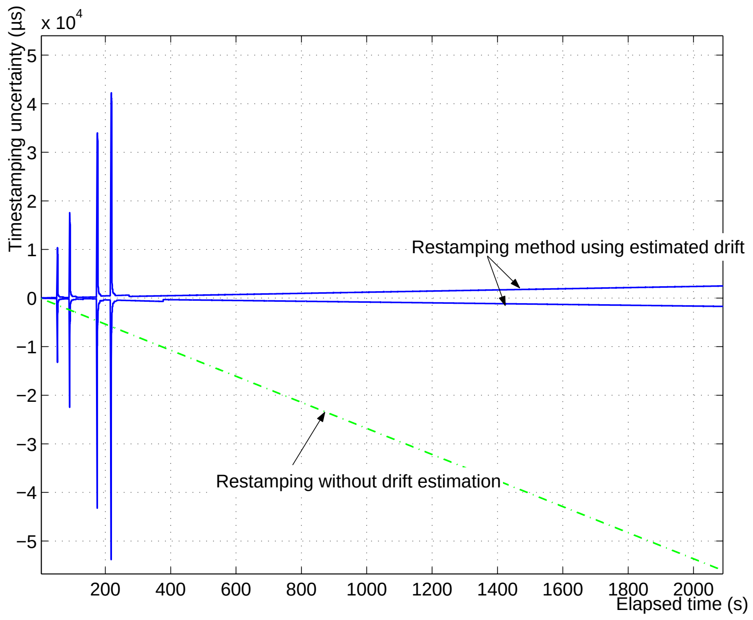


Figure 12: Timestamping interval width after a timestamp conversion

In Fig. 12, the X-axis is the elapsed time in seconds, and the Y-axis is the timestamping interval width. A translation is performed to set the initial timestamp to 0. When an interval date is used to represent the timestamp, the timestamp must be contained between the lower and upper bounds of the interval.

The timestamp conversions were performed both with and without drift estimations:

[1] *SCOOT-R* (for *Linux RTAI* or *Windows*) can be obtained by sending an e-mail to *paul.crubille@hds.utc.fr*. We plan to make it available under a *CeCILL* free license.

- timestamp conversion without taking drift into account (one dash-dot straight line, in the center of the graph, Fig. 12)
- timestamp conversion as described in this paper, with an estimated drift (two curved lines in the center of Fig. 12, representing the interval with respect to the bottom and top lines)

The first thing to be noticed is that a drift exists, so that a precise timestamp conversion cannot be performed without taking this drift into account (dash-dot straight line).
There are some increases in timestamping interval width, for example at approximately 60, 90, 170, and 220 seconds. These abrupt increases are due to a bus reset. The synchronous bus network clock time can change at these points, so the drift estimation is recomputed. Before a first drift estimation, the approximation value $[-120,\ 120]$ $\mu s/s$ is taken for the drift. Then, the drift interval diminishes progressively until it reaches the best estimation.

The interval width increases with time, because the drift interval width has an influence on the converted timestamp width: the greater the elapsed time between the clock time correspondence and the timestamp to convert, the worse the estimated timestamping width.

Other methods taking drift into account, such as NTP [Mills, 1991], can also be used to convert interval timestamping. However, these protocols need to exchange frequent messages to estimate the offsets between the various clocks, whereas the method proposed in this paper does not require explicit messages exchanges between the different computers.

Fig. 13 shows the increase in interval drift. The estimated drift interval uncertainty is 1 $\mu s/s$ in the best case, which corresponds to the time granularity. The value of 3 $\mu s/s$ is obtained after about 4 s, whereas 5 $\mu s/s$ is obtained after less than about 2.5 s.

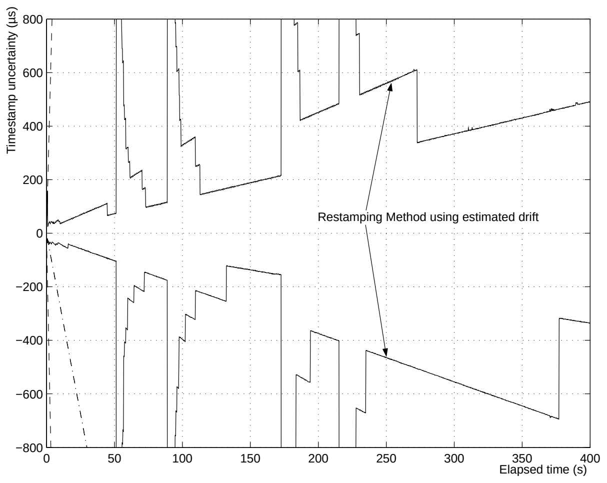


Figure 13: Timestamping uncertainty after a timestamp conversion: zoom on the timestamp conversion with the estimated drift between 0 and 400 seconds

In the optimum case, i.e., when the drift estimation is the best and t_c in equation (15) or (14) is close to t, the timestamp uncertainty is slightly more than 40 μs, as in the simulations, Sect. 4.2.1. Moreover, after 400 seconds, a timestamping interval width of about 800 μs is observed, which is about the same as the result simulated for theoretical performance evaluations, Fig. 8.

4.3.2 In-lab experimental results for post-processing timestamp synchronization

Some experiments were performed under *Linux LXRT* to validate the post-processing synchronization. They involved two computers and one camera connected via *FireWire*, see Fig. 14.

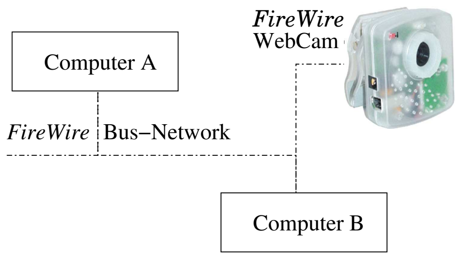


Figure 14: Experimental protocol for testing experimental post-processing conversion

Each computer has a different time and records the arrival time of each video frame, as shown in Fig. 15. In this figure we see that an image i is produced at the camera time t_i, and timestamped respectively at times $[t_{A_i}]$ and $[t_{B_i}]$ for computers A and B. As the images are transmitted at the same time on the synchronous bus-network, they are received at approximately the same time from computer A or computer B.

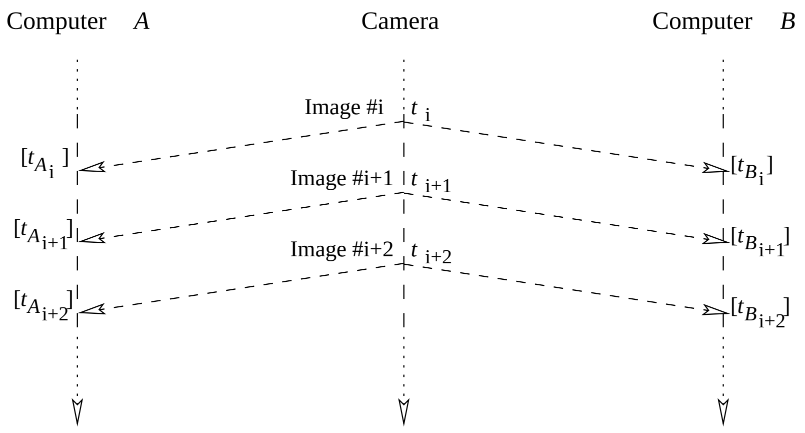
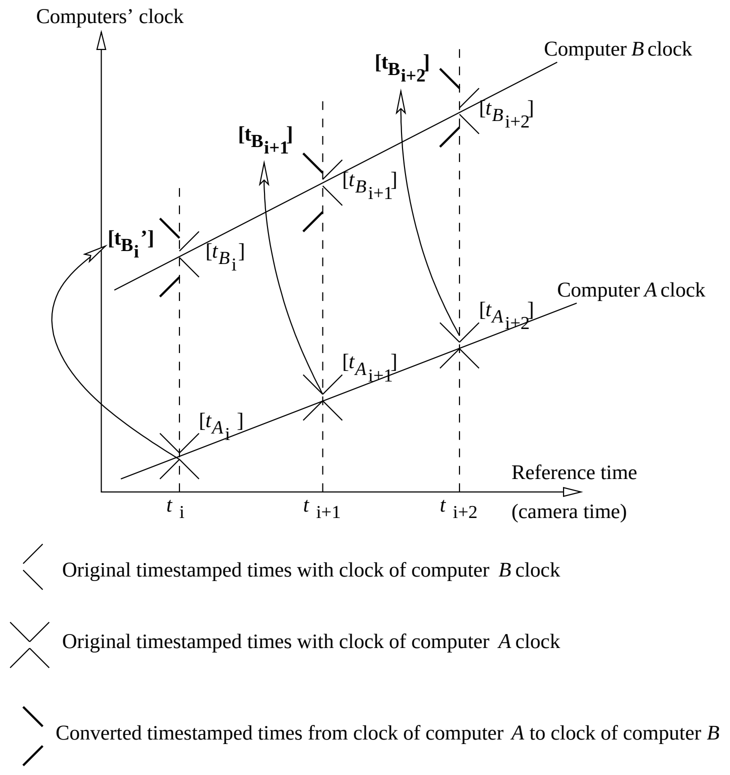


Figure 15: Camera and computers timestamping diagram

A post-processing synchronization is then computed in order to obtain image timestamps with the same reference time. As each computer receives each frame at the same time, the timestamps are the same in universal time, so that the validity of the post synchronization can be checked. Fig. 16 illustrates the conversion from computer A clock timestamps to computer B timestamps. Timestamps are known in the two reference times (from clock A and clock B), and are converted from computer A's to computer B's clock using our method.



Computers' clock
Computer B clock
Computer A clock
[tBi']
[tBi+1]
[tBi+2]
[tBi]
[tAi]
[tAi+1]
[tAi+2]
Reference time
(camera time)
t i
t i+1
t i+2
Original timestamped times with clock of computer B clock
Original timestamped times with clock of computer A clock
Converted timestamped times from clock of computer A to clock of computer B

Figure 16: Conversion of timestamps from computer A's clock to computer B's clock

A *player* was developed to visualize recorded images with their timestamps. When visualizing images with the computers' particular timestamps, the same images are not present at the same time: the computer timestamps are not synchronized. When visualizing images with timestamps converted to the same reference time, the same images are present on the two screens at the same time, as shown in Fig. 17: this shows that the same images are timestamped with the same timestamp.

Fig. 18 shows the timestamp synchronization errors in relation to the post-processing synchronization for an approximately 8-minute sequence. The timestamps for the reference computer time are set to 0, and the recomputed timestamps are set in relation to this 0. The correspondence computation was performed in two

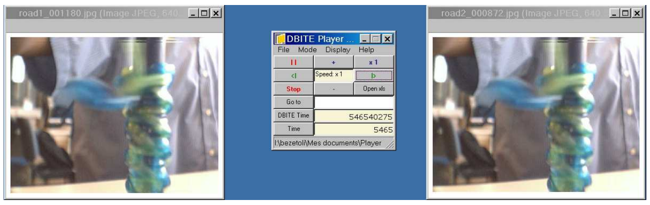


Figure 17: Restamped image-visualization screenshot

stages, because the drifts between the computers are not linear. The first sequence is from 0 s to about 200 s and the second from 200 s to the end.

This shows that the drift should be taken into account in certain cases, depending on the clocks: the experiments described were performed just after the computer boot, so components were not at their optimum temperature. A temperature variation implies a drift variation. The linearity of the drift can be checked by analyzing the *synchronization files*.

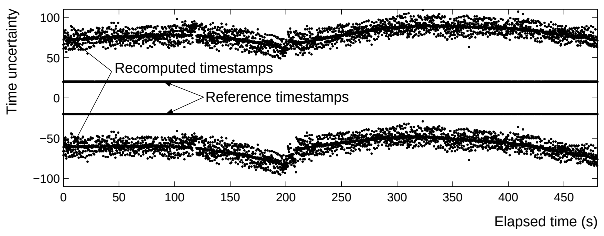


Figure 18: Timestamping interval width after a restamping computation

Several clocks were checked, generally showing that the drift becomes constant after about 15 minutes. Then, it is approximately constant, provided that the clock is subjected to a relatively constant temperature. In fact, oscillators, and consequently drifts, change with age and are affected by environmental variables, such as mechanical vibration, magnetic fields, and especially temperature.

Other longer experiments were also carried out, all of them showing that the converted timestamping interval encompassed the original interval.

Further experiments using a real system embedded in a vehicle were performed, including the on-line and post-processing timestamping processes described in this paper. These experiments were designed to test an *ADAS* (Advanced Driver Assistance Systems) function and formed part of the European *Roadsense* project [for Driving a Strategy that Evaluates Numerous SystEms, 2004], in collaboration with *Renault* [Bezet et al., 2006]. They lasted about 2 hours, during which time data were recorded.

Timestamping precision estimation can be useful for dynamic applications: e.g., a timestamp correction adjustment has been made for laser range scanner data, showing the influence of timestamping precision for dynamic applications using several sensors and computing resources such as robotics applications, vehicle and transport systems. The correction adjustment is very similar to computations shown in this paper, [Bezet and Cherfaoui, 2006].

5 Conclusions and future work

This paper has described on-line and post-processing timestamp conversion methods using interval date timestamping, and has presented theoretical and experimental results. These methods use comparisons with a common clock, the synchronous bus network clock, to perform precise on-line timestamp conversions and to avoid specific exchanges through the network.

The first advantage is that everyday hardware is used: *Firewire* is now often integrated within new laptops, for example. The timestamp conversion method presented here can be implemented with standard hardware, and little engineering input: the middleware used can be freely obtained and we plan to make it available as free software. The system developed is particularly suitable for fast prototyping of distributed applications and developing experimental applications using standard low-cost hardware. It does not require large and expensive systems which would be hard to implement. Estimating the timestamping uncertainty can be useful, particularly for dynamic systems. Then, if necessary, the experimental system can be implemented on dedicated systems, having better timestamping precision and better reliability.

A second advantage is that no clock is changed on any computer, which enables total parallelism, dynamic reconfigurations and a complete independence of computers, without the requirement that they all be connected together before the first timestamp conversion. Computers can therefore remain connected, and form independent networks able to connect together without any disturbance. Moreover, instantaneously precise on-line timestamping is possible, even after a network configuration change. Timestamping uncertainty makes it possible to evaluate precision, enabling timestamp values to be guaranteed.

Another advantage is that only a few simple calculations (interpolations) are required to perform the timestamp conversion. It is sufficient to compute the drift regularly and to record correspondences in a file. The conversion is direct: for on-line timestamping, very little memory is needed to store time equivalencies and to find the closest to the timestamp. For post-processing timestamp conversion, time equivalencies are stored in a file. The period of recorded time equivalencies can vary depending on precision requirements, remaining processor power and remaining capacity storage.

Last but not least, no exchanges are necessary among the different computers to maintain the clock correspondences. Time exchanges are only needed at data transmission: when a datum is sent, a field is added for the time. There are no exchanges limiting the number of connected computers: there can be as many computers as are required. One aim of this work is fast prototyping, where the optimum distribution of computation is not necessarily known in advance. With the proposed method, all computers can be connected, even if there are few data exchanges between computers.

Theoretical as well as practical experiments were performed to evaluate the performances and to validate the proposed timestamp conversion method, both types of experiment yielding approximately the same results, enabling us to predict the resulting timestamping interval widths depending on the application parameters, which include latencies, computer and network clock granularities and the drift between the different clocks.

Finally, applications including the proposed conversion methods have been implemented, and similar works have shown the influence of the timestamping precision for dynamic situations.

Several interesting perspectives follow from this work. First, a study of interval timestamping for data fusion should be done. This can be a problem when data are not all produced exactly synchronously. It might be interesting to study the consequences of interval date timestamping. Most algorithms do not accommodate interval timestamping. Interval dating takes account of timestamping uncertainty, leading to better data quality estimations.

In addition, the desired timestamp conversion might be used in other synchronous bus networks, e.g., the *Time Triggered Protocol, TTP* [Kopetz, 1997], where a network clock is available. However, some adaptations would be necessary. For example, although the *FireWire* clock is an interface clock, the *TTP* clock is given by an internal clock synchronization. This means that its drift is not exactly constant, but can vary slightly. In this case it may be supposed that the synchronous bus network clock drift is constant over a given time period which needs to be determined, and consequently a sliding window might be used to compute the drift.

Acknowledgement

The authors wish to acknowledge the financial support of the European FP5 RoadSense project (GRD1-2000-25572). They would also like to thank Philippe Bonnifait, Géry Brissot, Paul Crubillé and Amadou Gning for their support.

References

[Anderson, 1999] Anderson, D. (1999). *FireWire system architecture (2nd ed.): IEEE 1394a.* Addison-Wesley Longman Publishing Co., Inc.

[Bar-Shalom, 2000] Bar-Shalom, Y. (2000). Update with out-of-sequence measurements in tracking: exact solution. In *Proc. SPIE Vol. 4048, p. 541-556, Signal and Data Processing of Small Targets 2000, Oliver E. Drummond; Ed.*, pages 541–556.

[Bezet and Cherfaoui, 2005a] Bezet, O. and Cherfaoui, V. (2005a). On-line timestamping synchronization in distributed sensor architectures. In *Proc. of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2005)*, pages 396–404, San Francisco, California, USA. IEEE Computer Society.

[Bezet and Cherfaoui, 2005b] Bezet, O. and Cherfaoui, V. (2005b). Timestamping uncertainties in distributed data acquisition systems. In *Proc. of the 22nd IEEE Instrumentation and Measurement Technology Conference (IMTC 2005)*, OTTAWA, ONTARIO, CANADA.

[Bezet and Cherfaoui, 2006] Bezet, O. and Cherfaoui, V. (2006). Time error correction for laser range scanner data. In *Proc. of the Ninth International Conference of Information Fusion (FUSION 2006)*, Florence, Italy.

[Bezet et al., 2006] Bezet, O., Cherfaoui, V., and Bonnifait, P. (2006). A system for driver behavioral indicators processing and archiving. In *Proc. of the Ninth International IEEE Conference on Intelligent Transportation Systems - ITSC 2006*, Toronto.

[Blum et al., 2004] Blum, P., Meier, L., and Thiele, L. (2004). Improved interval-based clock synchronization in sensor networks. In *Proc. of the third international symposium on Information processing in sensor networks*, pages 349–358, Berkeley, California, USA. ACM Press.

[Chaaban et al., 2003] Chaaban, K., Crubillé, P., and Shawky, M. (2003). Scoot-r: a framework for distributed real-time applications. In *Proc. of the 7th int. conf. on princ. of dist. syst.*, La Martinique, France.

[Elson and Estrin, 2001] Elson, J. and Estrin, D. (2001). Time synchronization for wireless sensor networks. In *Proc. of the 15th International Parallel & Distributed Processing Symposium*, page 186. IEEE Computer Society.

[Elson et al., 2002] Elson, J., Girod, L., and Estrin, D. (2002). Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev.*, 36(SI):147–163.

[for Driving a Strategy that Evaluates Numerous SystEms, 2004] for Driving a Strategy that Evaluates Numerous SystEms, R. R. A. (2001-2004). http://www.eu-projects.com/roadsense/.

[Ganeriwal et al., 2003] Ganeriwal, S., Kumar, R., and Srivastava, M. B. (2003). Timing-sync protocol for sensor networks. In *Proc. of the first international conference on Embedded networked sensor systems*, pages 138–149, Los Angeles, California, USA. ACM Press.

[Hosek, 2005] Hosek, M. (2005). Clustered-architecture motion control system utilizing ieee 1394b communication network. In *Proc. of the 2005 American Control Conference (ACC 2005)*, volume 4, pages 2939– 2945, Portland, Oregon, USA.

[Jaulin et al., 2001] Jaulin, L., Kieffer, M., Didrit, O., and Éric Walter (2001). *Applied Interval Analysis.* Springer-Verlag.

[Kopetz, 1997] Kopetz, H. (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications*, volume 395. Kluwer Academic Publishers.

[Kopetz et al., 2006] Kopetz, H., Ademaj, A., and Hanzlik, A. (2006). Combination of clock-state and clock-rate correction in fault-tolerant distributed systems. *Real-Time Systems Journal, Volume 33, Numbers 1-3.*

[Marzullo and Owicki, 1983] Marzullo, K. and Owicki, S. (1983). Maintaining the time in a distributed system. In *Proc. of the second annual ACM symposium on Principles of distributed computing*, pages 295–305, Montreal, Quebec, Canada. ACM Press.

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

[Mills, 2003] Mills, D. L. (2003). A brief history of ntp time: memoirs of an internet timekeeper. *SIGCOMM Comput. Commun. Rev.*, 33(2):9–21.

[Obermaisser, 2004] Obermaisser, R. (2004). *Event-Triggered and Time-Triggered Control Paradigms.* Springer-Verlag Telos.

[PalChaudhuri et al., 2004] PalChaudhuri, S., Saha, A. K., and Johnson, D. B. (2004). Adaptive clock synchronization in sensor networks. In *Proc. of the third international symposium on Information processing in sensor networks*, pages 340–348, Berkeley, California, USA. ACM Press.

[Römer, 2001] Römer, K. (2001). Time synchronization in ad hoc networks. In *Proc. of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, pages 173–182, Long Beach, CA, USA. ACM Press.

[Schmid and Schossmaier, 1997] Schmid, U. and Schossmaier, K. (1997). Interval-based clock synchronization. *Real-Time Systems*, 12(2):173–228.

Dr Olivier BEZET received the M.S. degree in Computer Science, in 2002 and the Ph.D. degree in System and Information Technologies in 2005, from the University of Technology of Compiegne, France. He is currently a post-doctoral researcher in Computer Science at the Verimag Laboratory, Grenoble, France. His main research interest is now the Wireless Sensor Network area, its virtual prototyping, precise modeling and simulation.

Dr Véronique CHERFAOUI received the M.S. degree in Computer Science from the Lille University, France, in 1988 and the Ph.D. degree in control of systems from the University of Technology of Compiegne, France in 1992. She is an Associate Professor in computer engineering Department at the University of Technology of Compiegne. Her research interests in the Heudiasyc-CNRS laboratory are data fusion algorithms in distributed architecture, data association and real-time perception systems for intelligent vehicles.