Reliable and Self-Aware Clock: complete description

Andrea Bondavalli, Andrea Ceccarelli, Lorenzo Falai
University of Florence, Viale Morgagni 65, I-50134, Firenze, Italy
{ bondavalli, andrea.ceccarelli}@unifi.it, lorenzo.falai@resiltech.com

Abstract

This Technical Report provides a complete and exhaustive (even if preliminary) description of the Reliable and Self-Aware Clock (R&SAClock). the Reliable and Self-Aware Clock (R&SAClock) is a software component that allows to compute synchronization uncertainty, i.e. a conservative estimation on distance of a local clock from global time. The R&SAClock is a new software clock for resilient time information that provides both current time and current synchronization uncertainty, i.e. an estimation of the distance of local clock from an external global time. It is a low-intrusive component that hides to users the existence of both the synchronization mechanisms in use (possibly more than one) and the software clock.

1 Introduction

In many pervasive and distributed systems, it is required that nodes keep their clock synchronized with respect to a global time, that is a unique time view shared by the set of nodes involved. This is required to correctly execute applications and protocols. For example, many Wireless Sensor Networks (WSNs) distributed protocols for data sampling are characterized by real-time requirements, and all sensor nodes involved need to share a common view of time and operate to satisfy deadlines with respect to that shared view [1].

Typically the global time is realized using clock synchronization mechanisms. However, despite the use of such synchronization mechanisms, the time view that a local clock imposes to its node may deviate from global time. In fact distributed systems, and especially pervasive, open and adaptive systems are characterized by unpredictability and unreliability factors, because of changes in system dynamics, faults, or environmental changes [2], [3], [4]. Clock synchronization mechanisms, the clocks itself and consequently distance from global time are influenced by these characteristics: distance from global time may vary due to factors related both to the distributed system and network behavior and to the node internal behavior. Common causes to this variability due to the system behavior are unreliable networks and unpredictable communication delays, propagation delays, inaccessibility of the global time reference, network partitioning, changing environment (climate changes, electromagnetic interferences, ...), nodes failures. Variability can derive also from internal clock frequency changes (because of environmental factor or because of aging), or from any possible kind of failure in the clock synchronization algorithm or in the clocks and node itself [5].

Many systems that require synchronized clocks can take advantage from the awareness of actual distance from global time, or from a reliable estimation of such distance. Unfortunately, the actual distance is a variable factor very hard to predict, because of the previously stated problems. Worst case bounds on such distance are usually available, since they are imposed to systems as system requirements: unfortunately these bounds are far from typical execution scenarios and consequently are of practical little use. Instead the ability to furnish an adaptive conservative estimation on distance of local clock from global time, that we call *synchronization uncertainty*, could be useful in a large number of systems. A first discussion of the role of synchronization uncertainty in pervasive, open and adaptive real-time protocols and applications and in a set of typical message-passing activities has been presented in [6].

In this document we describe the *Reliable and Self-Aware Clock* (R&SAClock), a software component that allows to compute synchronization uncertainty. The R&SAClock is a new software clock for resilient time information that provides both current time and current synchronization uncertainty, i.e. an estimation of the distance of local clock from an external global time. It is a low-intrusive component that hides to users the existence of both the synchronization mechanisms in use (possibly more than one) and the software clock.

2 Time and clocks

In this Section we present a set of definitions and concepts that constitute a necessary background to provide a complete description of the problem and of the R&SAClock solution. In Table 1 we summarize the main definition we introduce in this Section.

2.1 Basic notions on time and clocks

Let us consider a distributed system, composed of n processes, or nodes. We define *global time* as the unique time view shared by the processes of the system, *perfect clock* as the clock that always fits the global time, and *global time node* as the node that owns the perfect clock. Given a clock c and any *time instant* (or simply *time* whenever clear from context) t, we define $c(t)$ as the *time value* (or simply *time* whenever clear from context) read by clock c at time t.

Every process of the distributed system has access to a local clock. The behavior of this local clock can be described defining three quantities, namely *precision*, *accuracy* and *drift*. Precision π describes how closely local clocks remain synchronized to each other at any time [5]. Accuracy α_i describes how the local clock i of the process P_i is synchronized at any time to an external global time [5]; accuracy is thus an upper bound to the distance between local clock and global time. Accuracy makes sense only in presence of external synchronization, in which an external absolute global time node is used as target of the synchronization. As a consequence of these definitions, in case of external synchronization considering for example a set of two nodes a and b respectively with accuracy α_a and α_b, precision is at least as good as $\pi = \alpha_a + \alpha_b$.

The *drift* $\rho_c(t)$ describes the rate of deviation of a clock c at time instant t from global time [5].

Clock synchronization is defined as the process of maintaining the properties of precision, accuracy and drift of a clocks set [5]; it is realized by appropriate *synchronization mechanisms*.

We define *offset* $\Theta_c(t)$ as the distance of local clock c from external global time at time t [7]. The offset is a variable that changes through time in a way hard to predict; its variations depend on the behavior and characteristics of local clock itself, local node, and whole distributed system and network. Accuracy α_c is such that, at any t, $\alpha_c \geq |t - c(t)| = \Theta_c(t)$; i.e., accuracy α_c is a bound sufficiently high to contain all possible offsets.

Figure 1 exemplifies the concepts of precision, accuracy, drift, offset and clock synchronization; the outside thick dashed lines represent the bound in the rate of drift, a fundamental assumption of clock synchronization mechanisms, since it allows to predict the maximum deviation after a given time interval.

2.2 From offset and accuracy to synchronization uncertainty

Despite their theoretical importance, usually accuracy α_c and offset $\Theta_c(t)$ are of practical little use for systems. In fact i) accuracy is usually a high value, far from offset, and it is not a representative estimation of current distance from global time, and ii) offset is difficult to measure (typically offset computation suffers of many sources of uncertainty, as unpredictable communication delays).

Synchronization mechanisms typically compute an *estimated offset* $\theta_c(t)$, but they usually offer no guarantees of closedness of this value with offset $\Theta_c(t)$.

These considerations, together with the consciousness that a clock is a *measurement* instrument, led us to define the *synchronization uncertainty* $U_c(t)$ as a conservative estimation of offset $\Theta_c(t)$ at time t: it always holds $\alpha_c \geq U_c(t) \geq \Theta_c(t)$. Synchronization uncertainty provides more precise and detailed information on distance from global time: in agreement with the typical definition of measurement uncertainty [8], we can identify synchronization uncertainty as an estimate of the degree of knowledge of the measurand (i.e., the global time), and it has to be included as part of the measurement result (i.e., it has to be estimated for each time value collected by the local clock).

These are the reasons why the Reliable and Self-Aware Clock has been developed: it is a new software clock that, at any time t, provides both time value $c(t)$ and synchronization uncertainty $U_c(t)$ (whenever clear from context, synchronization uncertainty will be called simply *uncertainty* in what follows). System elements that uses the R&SAClock have at their disposal the possibility to set requirements on distance from global time that have significant relevance.

3 The R&SAClock overview: services and specifications

In this Section we describe the R&SAClock component: we first present an high level overview of the R&SA-Clock, then we illustrate the specifications of the time values provided by of the R&SAClock to users and finally we show R&SAClock functional and non functional requirements.

Name	**Symbol**	**Description**
local clock (or *clock*)	$a, b, \ldots$	local clock
time instant (or *time*)	$t, t_1, \ldots$	time instant read from perfect clock
time value (or *time*)	$c(t)$	time value read from clock c at time instant t
precision	π	how closely local clocks remain synchronized to each other at time t
accuracy	α_c	upper bound to the distance between time of clock c and global time
drift	$\rho_c(t)$	rate of deviation of clock c from global time at time t
offset	$\Theta_c(t)$	distance of clock c from global time at time t
estimated offset	$\theta_c(t)$	estimation of $\Theta_c(t)$ performed by synchronization mechanism
synchronization uncertainty (or *uncertainty*)	$U_c(t)$	conservative estimation of $\Theta_c(t)$

Table 1. Summary of main concepts introduced in Section 2

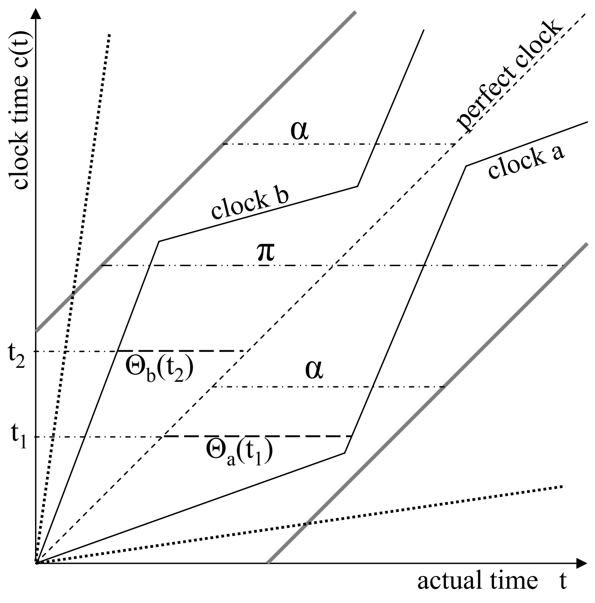


Figure 1. A set of two clocks: synchronization, accuracy, precision and drift

3.1 High level overview of main specifications and services

In Figure 2 we summarize a high level view of the R&SAClock component. System elements (that we will always call *users* in what follows) communicate with the R&SAClock by a user *interface*, and they query R&SAClock for current time and current uncertainty. When a user performs a query, the interface activates the *time value generator* algorithm, that creates an *enriched time value* (detailed in Section 3.2) in which current local time and current uncertainty are provided together. Then the enriched time value is delivered to the user. The time value generator algorithm creates the enriched time value using information provided by the local clock and by the *uncertainty evaluation* algorithm.

By gathering information from local clock, from synchronization mechanism, and from configuration parameters, the *uncertainty evaluation* algorithm estimates the uncertainty of the local clock. Moreover, this algorithm operates to maintain uncertainty within requirements set by users. In fact each user can impose an *accuracy requirement*, that is the worst uncertainty the user can accept to work properly; given a user A, we indicate its accuracy requirement as α_A. For any user A and any time t, the uncertainty evaluation algorithm operates to satisfy $U_c(t) \leq \alpha_A$, by forcing the synchronization mechanisms to perform synchronization attempts, if necessary. Special users may have the possibility to define configuration parameters of the uncertainty evaluation algorithm.

The *time value generator* algorithm and the *uncertainty evaluation* algorithm are detailed in Section 5.

The R&SAClock hides to users the existence of both the synchronization mechanisms in use (possibly more than one) and the clock; it is designed to work with any kind of synchronization mechanism and provides reliable information even in case the synchronization mechanisms in use are not working properly, or unfit conditions for synchronization happen (e.g. network becomes unavailable).

3.2 R&SAClock time specification: TAI timescale and the enriched time value

R&SAClock uses as global time the Temps Atomic International (TAI, [9]) timescale. The most widespread timescale is the Universal Time Coordinated (UTC, [9]), but in UTC a leap second is periodically inserted: the UTC time-growth function is monotonically increasing exception made for the leap second insertion action. Instead TAI has the same growth rate of UTC, and it is always monotonically increasing: this simplifies the development of time-related algorithms. Converting UTC data to TAI data and viceversa is low-effort [9], so R&SAClock can deal with synchronization mechanisms based on UTC without significant increase in computational or development costs.

R&SAClock delivers to users a new kind of time value, an *enriched time value*, which contains information on both local time and uncertainty. The enriched time value is the tuple

$$[likelyTAI; minTAI; maxTAI; FLAG]$$

At any time t and clock c, the value $likelyTAI$ is the time value $c(t)$ (i.e., $likelyTAI = c(t)$).

We now consider variables $minTAI$ and $maxTAI$: they represent the left and right synchronization uncertainty margin with respect to $likelyTAI$. More specifically, for a clock c at any time instant t, we extend the notion of synchronization uncertainty $U_c(t)$ distinguishing between a *right uncertainty* (positive) $U_{cr}(t)$ and a *left uncertainty* (negative) $U_{cl}(t)$ such that $U_c(t) = max\{U_{cr}(t), -U_{cl}(t)\}$. In case of *symmetric left and right uncertainty* with respect to $likelyTAI$ we have $-U_{cl}(t) = U_{cr}(t)$; in case of *asymmetric left and right uncertainty* with respect to $likelyTAI$ we have $-U_{cl}(t) \neq U_{cr}(t)$. In both cases of symmetric and asymmetric uncertainty, we have $minTAI = likelyTAI + U_{cl}(t)$ and $maxTAI = likelyTAI + U_{cr}(t)$.

The size of the interval $[minTAI; maxTAI]$ may vary, depending on computed synchronization uncertainty. The interval $[minTAI; maxTAI]$ is meant to include the time instant t and $likelyTAI = c(t)$.

In Figure 3 we summarize these observations: we depict both scenarios of asymmetric and symmetric uncertainty.

Finally, we describe the $FLAG$ field. It is a binary value: for any user A at any time t, the $FLAG$ value is 1 if uncertainty $U_c(t)$ satisfies the accuracy requirement α_A, and 0 otherwise (if no accuracy requirements have been established, $FLAG$ value is 0).

3.3 R&SAClock Requirements

In this Section we show the functional and non-functional requirements of R&SAClock.

3.3.1 Functional requirements

We divide functional requirements in i) interface requirements, ii) internal mechanisms requirements and iii) clock requirements.

About interface requirements, a user can set an accuracy requirement and can request an enriched time value. The user stops using R&SAClock by communicating that it quits the R&SAClock services.

Internal mechanisms requirements are defined as follows. R&SAClock must be able to compute synchronization uncertainty and to interact with the synchronization mechanisms (to collect data from the synchronization mechanisms and to force synchronization requests). It must be able to read configuration parameters for the set-up of the time value generator and uncertainty evaluation algorithms (special users may have the possibility to set or modify configuration parameters).

About clock requirements, R&SAClock must be able to incorporate a local clock, or to handle it. At least the ability to read the local clock must be provided. Since R&SAClock behaves as a new clock, it must satisfy typical clock requirements, as being an incremental timer that satisfies a determined accuracy and has specific drift bounds [10], [5].

3.3.2 Non-Functional requirements

The main non-functional requirement is that, given $likelyTAI = c(t)$, time t must be guaranteed to be within the interval $[minTAI, maxTAI]$ with a determined confidence. Discussions about the probability that t is within $[minTAI, maxTAI]$ will not be reported here.

Since R&SAClock behaves as a clock, it must satisfy the non-functional requirements of typical clocks. In particular, the procedure to create and deliver the enriched time value must have fast response time and high

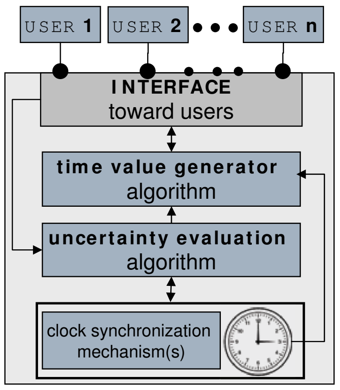

USER 1
USER 2
USER n
INTERFACE toward users
time value generator algorithm
uncertainty evaluation algorithm
clock synchronization mechanism(s)

Figure 2. A high level overview of the R&SAClock

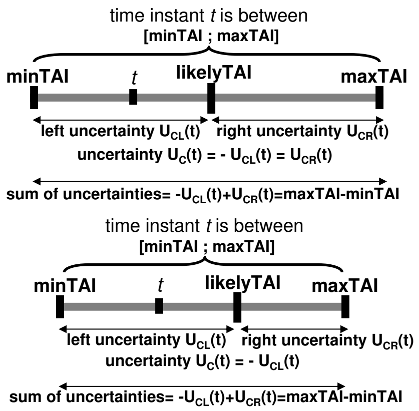

time instant t is between [minTAI ; maxTAI]
minTAI
t
likelyTAI
maxTAI
left uncertainty U_CL(t)
right uncertainty U_CR(t)
uncertainty U_C(t) = - U_CL(t) = U_CR(t)
sum of uncertainties= -U_CL(t)+U_CR(t)=maxTAI-minTAI
time instant t is between [minTAI ; maxTAI]
minTAI
t
likelyTAI
maxTAI
left uncertainty U_CL(t)
right uncertainty U_CR(t)
uncertainty U_C(t) = - U_CL(t)
sum of uncertainties= -U_CL(t)+U_CR(t)=maxTAI-minTAI

Figure 3. Enriched time value with symmetric and asymmetric uncertainty for clock c

throughput. The component should satisfy dependability-related requirements, mainly reliability and robustness [11]. Moreover, since R&SAClock is a clock and thus a measurement instrument, it should be low-intrusive, with low resource-consumption activities both in memory allocation and CPU usage [12], [8].

Two additional requirements are portability and usability. About portability, since R&SAClock is intended to be implemented on different kind of systems and synchronization mechanisms (and consequently, to be developed in different programming languages), its high level structure must be as general as possible, and it must allow different levels of interactions with clocks, synchronization mechanisms and users. About usability, the R&SAClock interface must be implemented in such a way that users can easily deal with R&SAClock.

4 Architectural design of the R&SAClock

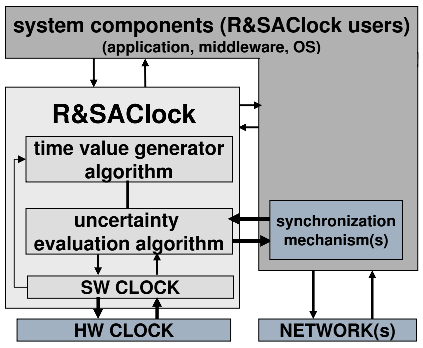
In this Section we detail the R&SAClock software architecture. We first present the R&SAClock location on system stack and its interactions with other system components, and then we detail R&SAClock in terms of its software structure.

4.1 R&SAClock architectural choices

The R&SAClock design and implementation choices may differ significantly depending on i) available synchronization mechanisms, ii) facilities that the synchronization mechanisms offer, iii) kind of node in use (in terms of computational power and operating system). However, in order to guarantee portability and easiness in adapting R&SAClock to different contexts, we restrict the architectural choices of the R&SAClock to two different scenarios. The two choices address systems that differ for the facilities that the synchronization mechanisms provide and for the possibilities offered to the developer to modify interactions with local clock and system components.

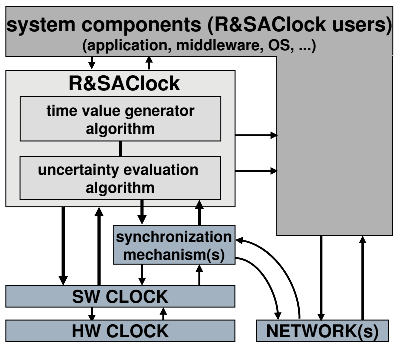
In the first scenario, depicted in Figure 4, R&SAClock wraps the software clock, and it acts as an intermediary between synchronization mechanisms and clock itself (it may instantiate and use his own software clock). In this scenario the software clock is completely handled by the R&SAClock. The software clock may be a pre-existent clock, or a new clock instantiated by the R&SAClock itself. Information from/to the synchronization mechanisms are used directly by the R&SAClock both to adjust the software clock and to execute the R&SAClock specific algorithms.

The second scenario is depicted in Figure 5. It is a lighter architectural solution for R&SAClock: R&SAClock lays on synchronization mechanisms and collects data from them and from the software clock, but it has only read-access to the software clock: the synchronization mechanisms completely handle it.



system components (R&SAClock users)
(application, middleware, OS)
R&SAClock
time value generator algorithm
uncertainty evaluation algorithm
synchronization mechanism(s)
SW CLOCK
HW CLOCK
NETWORK(s)

Figure 4. R&SAClock structured as a wrapper of software clock



system components (R&SAClock users)
(application, middleware, OS, ...)
R&SAClock
time value generator algorithm
uncertainty evaluation algorithm
synchronization mechanism(s)
SW CLOCK
HW CLOCK
NETWORK(s)

Figure 5. R&SAClock structured as layered on synchronization mechanism

We are currently developing an R&SAClock implementation that follows the schema of Figure 4 on the Lantronix-WiBox embedded system with a synchronization mechanism based on Global Positioning System (GPS, [13]) time-signalling technique in the car-to-car communication context [14]. The Lantronix-Wibox allows a high degree of interaction with the lower levels of the system stack, and the synchronization mechanism is not available; in this case the synchronization mechanism must be completely developed. This implementation will not be described here.

In Section 8 an implementation of R&SAClock following the schema proposed in Figure 5 has been developed on the Linux operating systems with the NTP synchronization mechanism: NTP provides detailed information on synchronization and a large set of functionalities, so R&SAClock can use NTP functionalities and data.

4.2 R&SAClock software structure

In this Section we describe the software structure of the R&SAClock. In Figure 6 we represent a sample schema of the R&SAClock implementation inheritance tree. First, the various versions of the R&SAClock are differentiated by the kind of synchronization mechanism in use, and then (at a lower level) for the kind of operating system: in fact typically a synchronization mechanism provides similar functionalities and behavior (and consequently similar implementations of the R&SAClock internal algorithms) on different kinds of systems.

Regarding software objects, we identify four software objects, developed to satisfy functional requirements shown in Section 3.3. These elements are shown in Figure 7, and are the following: *R&SAClock_Starter*, *Main_Core*, *User_Interace* and *Clock*.

The *R&SAClock_Starter* activates the R&SAClock, executing the function *startR&SAClock*.

The *User_Interface* allows users to interact with the R&SAClock. Users get the enriched time values by function *getTime*. Special users may have privileged access to configure execution parameters (function *setConfigValues* and *getConfigValues*). Users have the possibility to set their own accuracy requirements by *setAccuracyRe-*

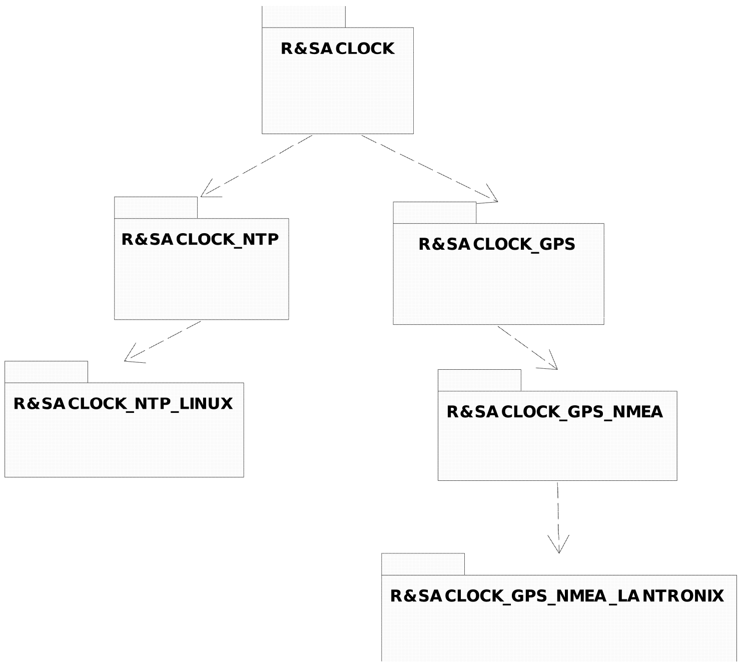


Figure 6. R&SAClock inheritance-tree

quirement function. The interactions between users and *User_Interface* may be implemented in different ways (e.g. sockets, function call, ...) depending on the kind of system in use. Users stop using R&SAClock services by a *quit* function.

The *Main_Core* interacts with both the *User_Interace* and the *Clock*. It contains the uncertainty evaluation algorithm and the time value generator algorithm, it processes data received by users (it collects the accuracy requirements by function *setAccuracyRequirement*), it maintains the main execution parameters (by function *storeConfigValues*), and it commands read/write actions to the *Clock*. It communicates with the synchronization mechanism to achieve needed data by function *getSynchrData* and it forces synchronization attempts whenever necessary by function *forceSynchronization*. Synchronization uncertainty is computed by function *computeUncertainty*; function *sleep* computes the time period in which we have guarantees that synchronization uncertainty is within the accuracy requirements set by users.

The *Clock* is connected to the *Main_Core*; in the first architectural scenario previously described (see Figure 4), it encapsulates the software clock or it maintains its own software clock; in the second architectural scenario (see Figure 5), it is an interface towards the software clock. It contains functions to read and write the software clock (respectively, functions *getClockTime* and *writeClock*).

5 Internals of a R&SAClock

We focus now on the description of the R&SAClock internal algorithms that provide the time services: the *time value generator* algorithm and the *uncertainty evaluation* algorithm. As described in Section 3, the time value generator algorithm generates the *enriched time value* (whenever it is requested by users) in which local time and synchronization uncertainty are provided together, while the uncertainty evaluation algorithm computes synchronization uncertainty and tries to keep it within accuracy requirement α_A for each user A.

In this Section we depict the algorithms without focusing on implementations of the functions used, in particular of functions *getSynchrData*, *forceSynchronization*, *computeUncertainty* and *sleep*. The reason is that these functions can be implemented in several different ways depending on i) system assumptions, ii) kind of synchronization mechanisms in use and ii) required degree of confidence that t is within the interval $[minTAI, maxTAI]$. Instead in Section 6 we will specify a set of assumptions and then we will present a possible implementation of the functions for systems that satisfy the set of assumptions.

For clarity, in Table 1 and Table 2 we summarize the main quantities and parameters used in this Section.

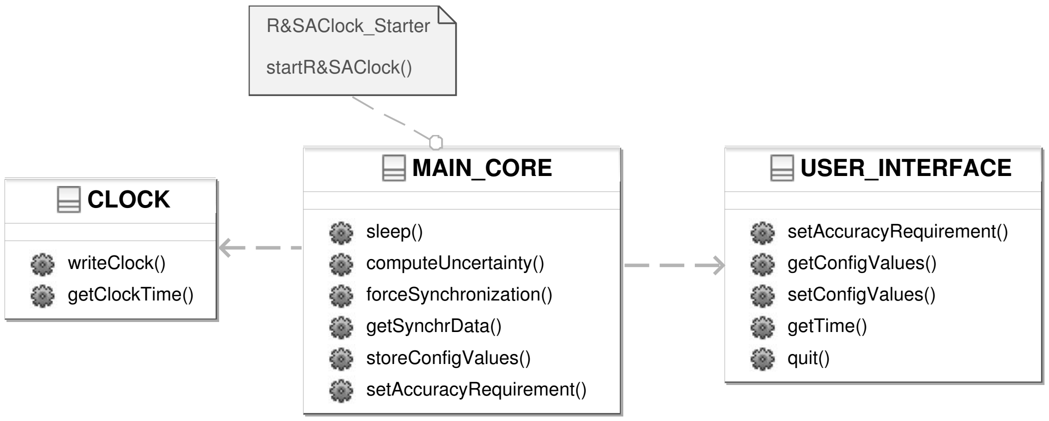


Figure 7. Overview of R&SAClock software objects

5.1 Assumptions on system in use

For simplicity, we consider a system with symmetric left and right uncertainty with respect to local time $likelyTAI$, that implies $U_c(t) = U_{cr}(t) = -U_{cl}(t)$ (see Section 3.2); however the algorithms can be easily adapted to situations of asymmetric uncertainty. For simplicity we assume only one synchronization mechanism on the system.

5.2 Time value generator algorithm

The time value generator algorithm creates the enriched time value $[minTAI, maxTAI, likelyTAI, FLAG]$. In Figure 8 we present the pseudocode of the algorithm. In case the function *computeUncertainty* needs input parameters, these parameters are stored in the uncertainty evaluation algorithm.

Let us consider a local clock c, a time instant t and a user A that requests the enriched time value. It is easy to note that $likelyTAI$ is collected querying the local clock (since $likelyTAI = c(t)$).

To compute $minTAI$ and $maxTAI$, it is required to compute left uncertainty $U_{cl}(t)$ and right uncertainty $U_{cr}(t)$. The computation of left and right uncertainty is obtained executing function *computeUncertainty*. Given $U_{cr}(t) =$ *computeUncertainty()*, $maxTAI$ is computed as $likelyTAI + U_{cr}(t)$. Since at any t we have $U_{cl}(t) = -U_{cr}(t)$, $minTAI$ is computed too.

Acronym	**Name and description**
$U_{cl}(t)$ and $U_{cr}(t)$	left and right uncertainty for clock c at time t
$U_c(t)$	for clock c at time t, $\max\{U_{cr}(t), -U_{cl}(t)\}$
α_A	accuracy requirement of user A: worst synchronization uncertainty A can accept
$TSLPC$	counter that represent "Time elapsed Since Last (more recent) Parameters Computation"
$\alpha LIST$	accuracy list: ordered list where accuracy requirements of users are stored
α_{START}	the default accuracy requirement at algorithm start-up
α_{FIRST}	the first value of the accuracy list (the smallest value of the list)
$sleepTime$	time period in which uncertainty is surely not greater than α_{FIRST}

Table 2. Main quantities and parameters involved in Section 5

```
receive enriched time value request from user A
likelyTAI = time of local clock
right uncertainty = computeUncertainty( )
maxTAI= likelyTAI + right uncertainty
minTAI= likelyTAI - right uncertainty

if (αA > right uncertainty)
   FLAG = 0
else
   FLAG = 1

return [likelyTAI, minTAI, maxTAI, FLAG]
```

Figure 8. Time value generator algorithm

Finally, the value $FLAG$ for user A is 1 if accuracy requirement α_A is not smaller than uncertainty $U_{cr}(t)$ (i.e., $\alpha_a \geq U_{cr}(t)$), or no accuracy requirements have been set for A; otherwise $FLAG$ value is 0. The enriched time value is now set.

5.3 The uncertainty evaluation algorithm

In this Section we detail the *uncertainty evaluation* algorithm. This algorithm equips R&SAClock with the ability to compute uncertainty; it is executing during all life-time of the R&SAClock, and the enriched time value created by the time value generator algorithm depends on data provided by this algorithm. For simplicity, we subdivide the algorithm in three phases: set-up phase, iterative phase and sleep phase. In Figure 9 we present the pseudocode of the algorithm.

Let us consider a local clock c at any time t. As for the time value generator algorithm, we focus on the

```
Initialize parameters: storeConfigValues( )
Thread 1: evaluate uncertainty
while(true)
    synchronization values = getSynchrData( )
    start chronometer TSLPC=0
    right uncertainty= computeUncertainty( )
    if(right uncertainty > αFIRST)
        forceSynchronization( )
        synchronization values = getSynchrData( )
        restart chronometer TSLPC
        right uncertainty= computeUncertainty( )
        while(right uncertainty > αFIRST)
            wait(predetermined time interval)
            forceSynchronization( )
            synchronization values = getSynchrData( )
            restart chronometer TSLPC
            right uncertainty= computeUncertainty( )
        endWhile
    endIf
    sleepTime = sleep( )
    wait(sleepTime)
endWhile
Thread 2: update αList
while(true)
    wait for input from users
    if(setAccuracy(αA) received from user A)
        add αA to αLIST
        if(setAccuracy(αA) received from user A)
            if(αA is the new αFIRST and Thread 1 sleeps)
                break sleep of Thread 1
    if(quit() received from user A)
        delete αA from αLIST and update αFIRST
endWhile
```

Figure 9. Uncertainty evaluation algorithm

computation of right synchronization uncertainty $U_{cr}(t)$.

On a configuration file there are start-up parameters i) specific of the four critical functions enlisted at the beginning of the Section and ii) α_{START}. The parameter α_{START} represents a starting accuracy requirement. In absence of accuracy requirements set by users, R&SAClock will operate to maintain uncertainty not greater than α_{START}.

Phase 1: set-up. At the algorithm start-up, a set-up phase is performed. Parameter α_{START} is read from configuration file (function *storeConfigValue*) and the ordered list $\alpha LIST$ is instantiated. This list contains all the accuracy requirements provided by users: we call the first parameter of the list (the closest to zero) α_{FIRST}. At algorithm start-up, α_{FIRST} is set equal to α_{START}; during its whole execution, the algorithm receives accuracy requirements from users and stores their values in the $\alpha LIST$; when a user quits the R&SAClock services, its value is removed from the $\alpha LIST$. Note that α_{FIRST} may need to be updated whenever the list is changed.

Let us suppose that function *getSynchrData* is executed at time t_1 (as a consequence, fresh information provided by synchronization mechanism are available at time t_1). Then $U_{cr}(t_1)$ is computed by function *computeUncertainty* and the time value $c(t_1)$ is collected. When such information is obtained (in some synchronization mechanisms, eg. [7], it may be necessary to wait an amount of time before *getSynchrData* is completed), the value "Time elapsed Since Last Parameters Computation" $TSLPC$ is initialized. The chronometer $TSLPC$ represents the time period elapsed since last computation of synchronization values has been performed by the synchronization mechanism. The set-up phase of the algorithm is now completed and Phase 2 is started.

Phase 2: Iterative phase. In this phase, the algorithm operates to have right uncertainty not greater than α_{FIRST}. At any t, if right uncertainty is greater than the threshold α_{FIRST} (i.e., $U_{cr}(t) > \alpha_{FIRST}$), a synchronization attempt (function *forceSynchronization*) is forced. When new fresh information from the synchronization mechanism is available (suppose at time $t_2 > t$), the counter $TSLPC$ is restarted and right uncertainty $U_{cr}(t_2)$ is computed. If right uncertainty is still greater than α_{FIRST} (i.e., $U_{cr}(t_2) > \alpha_{FIRST}$), synchronization attempts are forced at periodic time intervals (the time intervals must be long enough to assure that the algorithm is not resource-consuming), until right uncertainty is not greater than α_{FIRST}.

On the other hand, if at any time instant right uncertainty is computed and it is not greater than α_{FIRST} (for example, at time t we could have that $U_{cr}(t) \leq \alpha_{FIRTS}$), then there is no need to ask for synchronization attempts. Phase 3 is started.

Phase 3: Sleep phase. The uncertainty evaluation algorithm waits to recompute right uncertainty for a $sleepTime$ time period (this period is computed by the *sleep* function). The $sleepTime$ identifies the time in which right uncertainty, that is supposed to increase as counter $TSLPC$ increases, is surely not greater than α_{FIRST}. When $sleepTime$ period expires, it means that our computed right uncertainty is now equal to α_{FIRST}: this implies that there is the *potential risk* that offset is equal to the accuracy requirement α_{FIRST}. So fresh synchronization data are collected, right synchronization uncertainty is computed, and Phase 2 is started.

Finally, note that the $sleepTime$ counter expires whenever a new α_{FIRST} smaller than the previous one is set.

6 A possible implementation of main time-related functions

The functions *forceSynchronization*, *getSynchrData*, *computeUncertainty* and *sleep* are critical to the execution of the R&SAClock: however, there is not an univocal way to implement them. While implementation of functions *forceSynchronization* and *getSynchrData* strictly depends on the facilities that the synchronization mechanism offers, the implementation of the other two functions varies depending on i) synchronization mechanism, ii) system assumptions, and ii) required degree of confidence that t is within the interval $[minTAI, maxTAI]$. Moreover, different theoretical approaches can bring to different implementations.

In this Section we present a possible implementation of functions *computeUncertainty* and *sleep*. A theoretical proof to show that the implementation is correct, i.e. the functions behave as expected and consequently the uncertainty evaluation and the time value generator algorithms hold, is reported in Section 7: in that section it is shown that if system assumptions of Section 6.1 hold, then time instant t and time value $c(t) = likelyTAI$ are guaranteed to be within the computed interval $[minTAI, maxTAI]$.

For clarity, in Table 1, Table 2 and Table 3 we summarize the main quantities and parameters used in this Section.

6.1 Assumptions on system and synchronization mechanism in use

We consider the same assumptions of Section 5.1. Moreover, we assume that the synchronization mechanism in use allows collecting the following two values: the root delay $RD(t)$ and the estimated offset $\theta_c(t)$ (function *getSynchrData* collects these values).

We define *root delay* as the time interval (observed by the local node) needed to exchange a message between local node and global time node. When distributed nodes that executes the synchronization mechanism are organized in a tree-like structure (e.g. NTP), the global time node is the root node of the tree. In some synchronization mechanisms (e.g. NTP) the root delay of a node at level i is computed as the round trip delay of a packet sent up to the root of the tree and then received back to the node. In other synchronization mechanisms (e.g. GPS, or Reference Broadcast Synchronization [15]), root delay is the one-way transmission delay between the global time node and the local node (with no intermediary in GPS, and with possible intermediaries in RBS). Then, by root delay $RD(t)$ we mean the root delay computed at time t.

Value $\theta_c(t)$ is instead the estimated offset collected by the synchronization mechanism at time t.

We assume that estimated offset and root delay are computed simultaneously by the synchronization mechanism, and that the R&SAClock, interacting with the synchronization mechanism, is able to get the two parameters and the time value at which they are computed.

6.2 Function *computeUncertainty*

Since $U_{cr}(t) = -U_{cl}(t)$ by previous assumption, *computeUncertainty* computes only right uncertainty $U_{cr}(t)$.

To calculate $U_{cr}(t)$, we need to introduce the drift bound σ_c. This parameter is a positive constant that represents an upper bound on drift $\rho_c(t)$ at any time t. We assume that at any t it holds $\rho_c(t) \leq \sigma_c < 1$: the value 1 means that drift bound is lower than "1 second per second" (we believe it is a reasonable assumption). The σ_c may be read by configuration file or set by special users. Its value is maintained by the uncertainty evaluation algorithm.

The three values i) estimated offset $\theta_c(t)$, ii) root delay $RD(t)$, and iii) "Time elapsed Since Last Parameters Computation" $TSLPC$ are required: these values are maintained and possibly updated in the uncertainty evaluation algorithm. Supposing the last computation of estimated offset and root delay has been performed at time t_1,

Acronym	Name and description
RD	root delay: transmission delay (one-way or round trip, depending on synchronization mechanism) from local node to global time node (including all system-related delays)
Θ	estimated offset: best effort estimation of distance from actual time
Θ_T	True, actual offset: actual distance from actual time
$RDoptimum$	root delay optimum: the smallest root delay that can be achieved
σ_c	drift bound: upper bound on drift for clock c, i.e. $\sigma_c \geq \rho_c(t)$ for any t
γ	time delay due to system intrusiveness at global time node
δ	round trip delay between local and global time node, excluding system delays of global time node, measured with two timestamp of the local node and two of the global time node
T_{AB}	actual one-way delay from local node A to global time node B
T_{BA}	actual one-way delay from global time node B to local node A

Table 3. Main quantities and parameters involved in Section 6

at any $t \geq t_1$ we set $TSLPC = \frac{c(t)-c(t_1)}{1-\sigma_c} \geq t - t_1$ (in fact we cannot guarantee that $c(t) - c(t_1) = t - t_1$): a proof that in the assumptions of Section 6.1 this inequality holds can be found in Section 7.

Given t_1 as before, at any $t \geq t_1$ right uncertainty $U_{cr}(t)$ is computed by the following equation:

$$U_{cr}(t) = |\theta_c(t_1)| + RD(t_1) + (\sigma_c \cdot TSLPC) \tag{1}$$

Note that all values in the equation are positive. The last part of the equation states that, since last update of synchronization parameters, right uncertainty $U_{cr}(t)$ has been incremented at a rate higher than drift of local clock (since $\sigma_c \geq \rho_c$, and $TSLPC \geq t - t_1$). The basic idea of the equation is that given at time t_1 a right uncertainty that is no smaller than offset (i.e., $U_{cr}(t_1) \geq \Theta_c(t_1)$), then right uncertainty will still be no smaller than offset at time $t \geq t_1$.

In some cases, Equation 1 may require the additional contribution of local system intrusiveness and local clock resolution. Finally, if the optimum root delay $RDoptimum$ can be estimated, then Equation 1 can be rewritten as $U_{cr}(t) = |\theta_c(t_1)| + (RD(t_1) - RDoptimum) + (\sigma_c \cdot TSLPC)$.

A theoretical proof that, in the assumptions of Section 6.1, at any t it holds $U_{cr}(t) \geq \Theta(t)$ (i.e., R&SAClock requirements are never violated) can be found in [16].

6.3 Function *sleep*

The $sleepTime$ value is set equal to $sleepTime = (\alpha_{FIRST} - U_{cr}(t))/\sigma_c$. This counter identifies the time interval that the right uncertainty $U_{cr}(t)$, assuming a drift equal to σ_c, requires to become equal to α_{FIRST}. Note that $sleepTime$ is always positive by obvious assumptions on involved parameters. A theoretical proof that in the assumptions of Section 6.1 $sleepTime$ is computed in such a way that it never happens $\Theta(t) > \alpha_{FIRST}$ (that is, R&SAClock requirements are never violated) can be found in [16].

7 Correctness of functions *computeUncertainty* and *sleep*

In this section a proof of correctness of the two functions *computeUncertainty* and *sleep* is shown.

In Figure 10 we summarize the most critical variables of the proofs. Let A be the local node, and B the global time node. To simplify the notation, let Θ_c (instead of $\Theta_c(t_4)$) and θ_c (instead of $\theta_c(t_4)$) represent respectively the offset and estimated offset computed in figure; time value T_1 and T_4 are collected reading clock of node A at time instant t_1 and t_4 (we write T_1 and T_4 instead of $A(t_1)$ and $A(t_4)$), while T_2 and T_3 are collected reading clock of node B at time instant t_2 and t_3 (we write T_2 and T_3 instead of $B(t_2)$ and $B(t_3)$).

Let m be a message sent by A to B, with one-way duration $a = T_2 - T_1$, and then returned from B to A with one-way duration $b = T_3 - T_4$.

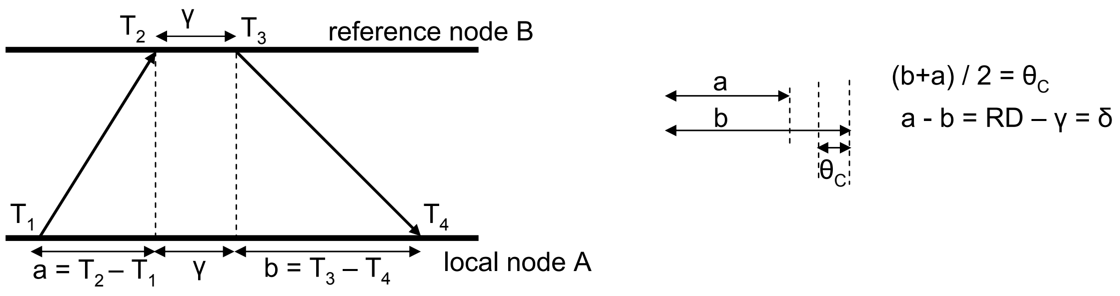


Figure 10. Main parameters of theoretical proof

The estimated offset is computed as $\theta_c = ((T_2 - T_1) + (T_3 - T_4))/2 = (a + b)/2$ (as in [7]).

The round-trip delay between A and B is the root delay $RD(T_4) = T_4 - T_1$. Time interval $\gamma = T_3 - T_2$ is the message processing time on node B. The observed round-trip transmission time δ is $\delta = a - b \leq |a| + |b| + \gamma = RD(T_4)$.

We start the proof showing that:

Theorem 1. *In the assumptions of Section 6.1, it holds* $|\theta_c| + RD(T_4) \geq |\Theta_c|$.

Proof. In what follows, we show that right uncertainty $U_{cr}(T_4)$ is such that $U_{cr}(T_4) \geq |\Theta_c|$. Let T_{AB} be the actual transmission delay of a message from A to B, we can state that $T_{AB} = a - \Theta_c = (T_2 - T_1) - \Theta_c \geq 0$, and thus $a \geq \Theta_c$; in a similar way, the actual transmission delay T_{BA} of a message from B to A is $T_{BA} = \Theta_c - b \geq 0$ and then $b \leq \Theta_c$. It follows that $b \leq \Theta_c \leq a$.

We then have $b = \frac{a+b}{2} - \frac{a-b}{2} \leq \Theta_c \leq \frac{a+b}{2} + \frac{a-b}{2} = a$, and substituting we have $\theta_c - \frac{\delta}{2} \leq \Theta_c \leq \theta_c + \frac{\delta}{2}$.

The root delay RD is such that $RD(T_4) = |T_4 - T_1| \geq a - b \geq \frac{\delta}{2}$: it includes the unpredictable message processing time on node B and possible asymmetric behavior of the round trip packet exchange.

Then $|\theta_c| + RD(T_4) \geq |\theta_c| + \frac{|\delta|}{2} \geq |\Theta_c|$. □

We show now that the following property of the $TSLPC$ value is true:

Theorem 2. *In the assumptions of Section 6.1, given a time instant* t_1 *and offset* $\Theta_c(t_1)$, $\forall\ t \geq t_1$ *it follows* $TSLPC = \frac{c(t)-c(t_1)}{1-\sigma_c} \geq t - t_1$

Proof. $\forall\ t_2 \in [t_1, t],\ \ \sigma_c \geq \rho_c(t_2)$. Since $t_1 = c(t_1) + \Theta_c(t_1)$ and $t \leq t_1 + \sigma_c \cdot (t - t_1) = c(t_1) + \Theta_c(t_1) + \sigma_c \cdot (t - t_1)$, then $t - t_1 \leq c(t) - c(t_1) + \sigma_c \cdot (t - t_1)$, from which $TSLPC = \frac{c(t)-c(t_1)}{1-\sigma_c} \geq t - t_1$. □

We can finally show that Equation 1 guarantees that absolute offset $|\Theta_c(t)|$ is smaller than synchronization uncertainty:

Theorem 3. *In the assumptions of Section 6.1,* $\forall\ t \geq t_4$, *it holds* $U_{cr}(t) = |\theta_c| + RD(T_4) + (\sigma_c \cdot TSLPC) \geq \Theta_c(t)$

Proof. If $t = t_4$, it follows from Theorem 1.

If $t > t_4$, it is sufficient to note that, since $\forall\ t_2 \in [T_4, t]$ we have $\sigma_c \geq \rho_c(t_2)$, then $\sigma_c \cdot (t - T_4)$ assures that synchronization uncertainty has a rate of deviation from global time greater than the local clock, no matter how much time has passed since last synchronization action. Finally, from Theorem 2, $TSLPC \geq (t - T_4)$: this ends the proof. □

Regarding Theorem 3, we considered the whole root delay RD and not $RD/2$ (as Theorem 1 could suggest), in order to include in the formula all possible sources of measurement uncertainties and delays, and to be in agreement with possible synchronization algorithms that assume $T_2 \approx T_3$ (thus not considering γ). In some cases, Equation 1 may require the additional contribution of local system intrusiveness, and local clock resolution. Finally, if the optimum root delay $RDoptimum$ can be estimated, then Equation 1 can be rewritten as $U_{cr}(t) = |\theta_c(t_1)| + (RD(t_1) - RDoptimum) + (\sigma_c \cdot TSLPC)$.

Finally, we prove the correctness of the *sleep* function by the following theorem:

Theorem 4. *In the assumptions of Section 6.1, given* $sleepTime = (\alpha_{FIRST} - U_{cr}(T_4))/\sigma_c$, *then* $\forall\ t \in [t_4, t_4 + sleepTime]$ *it never happens* $\Theta(t) > \alpha_{FIRST}$

Proof. If $\alpha_{FIRST} = U_{cr}(t)$, then $sleepTime = 0$ and the theorem follows trivially. If $\alpha_{FIRST} > U_{cr}(t)$, it follows from the second part of Theorem 3. □

8 Implementation for NTP synchronization mechanism and Linux OS

In this Section we briefly present a C++ implementation of the R&SAClock for NTP synchronization mechanism and Linux OS; a more detailed description can be found in [14].

First of all, we stress that the combination NTP - Linux is such that the system assumptions presented in Section 5 and Section 6 hold: consequently we can implement the R&SAClock following solutions proposed in those Sections.

NTP uses the UTC timescale; however, UTC-TAI conversion is simple, and NTP itself provides information to convert time values from UTC timescale to TAI timescale [9]. Moreover, NTP provides a large set of functionalities and data, and directly manages the software clock: thus the software architectural choice presented in Figure 5 is chosen, giving to the R&SAClock read-only access to the local clock.

When NTP performs synchronization attempts, it updates contemporaneously root delay and estimated offset and it writes the collected values on a log file. NTP performs synchronization attempts at periodic time intervals, but it is not possible for an external component to force synchronization attempts: the function *forceSynchronization* is not implemented, and thus R&SAClock ability of keeping synchronization uncertainty within accuracy requirements is currently not implemented. Moreover, R&SAClock can not ask NTP to refresh root delay and estimated offset.

Because of previous observations, the uncertainty evaluation algorithm is realized in the following way: R&SAClock listens on the NTP log file by function *getSynchrData*, and whenever R&SAClock detects that NTP updated the log file, it records the estimated offset and the root delay and restarts the $TSLPC$ counter. Without the need to compute a $sleep$ time, the function *getSynchrData* is invoked again.

Regarding the time value generator algorithm, function *computeUncertainty* is implemented as stated in Section 6; local clock resolution (1 microsecond in our system) is added to Equation 1, while contribution of system intrusiveness is considered negligible.

9 Conclusions

In this technical report we present a complete description of the Reliable and Self-Aware Clock. The Reliable and Self-Aware Clock (R&SAClock) is a new software clock for resilient time information that provides both current time and current synchronization uncertainty; moreover it pairs time-related information with an alarm event that makes R&SAClock users aware of poor synchronization. While hiding to users the existence of both the synchronization mechanisms in use and the software clock, it interacts with the synchronization mechanisms to request synchronization actions in order to keep synchronization uncertainty within bounds set by users.

Acknowledgements

This work has been partially supported by the European Community through the project IST-FP6-STREP-26979 (HIDENETS - HIghly DEpendable ip-based NETworks and Services).

We would like to thank Stefano Brilli for his work in implementing a C++ version of the R&SAClock for NTP-Linux.

References

[1] Deepak Ganesan, Sylvia Ratnasamy, Hanbiao Wang, and Deborah Estrin. Coping with irregular spatio-temporal sampling in sensor networks. *SIGCOMM Comput. Commun. Rev.*, 34(1):125–130, 2004.

[2] Kay Römer, Philipp Blum, and Lennart Meier. Time synchronization and calibration in wireless sensor networks. In Ivan Stojmenovic, editor, *Handbook of Sensor Networks: Algorithms and Architectures*, pages 199–237. John Wiley & Sons, September 2005.

[3] M. Satyanarayanan and et al. Pervasive computing: Vision and challenges. *Personal Communications, IEEE*, 8(4):pp. 10–17, August 2001.

[4] C. Almeida and P. Veríssimo. Using the timely computing base for dependable qos adaptation. In *20th IEEE Symposium on Reliable Distributed Systems (SRDS'01)*, 2001.

[5] P. Verissimo and L. Rodriguez. *Distributed Systems for System Architects*. Kluwer Academic Publisher, 2001.

[6] Andrea Bondavalli, Andrea Ceccarelli, and Lorenzo Falai. A self-aware clock for pervasive computing systems. In *PDP '07: Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, Washington, DC, USA, 2007. IEEE Computer Society.

[7] D.L. Mills. Internet time synchronization: the network time protocol. *IEEE Trans. Communications 39, 10*, pages 1482–1493, 1991.

[8] BIMP, IEC, IFCC, ISO, IUPAC, and OIML. *Guide to the expression of uncertainty in measurement*, 1993.

[9] Judah Levine and David Mills. Using the network time protocol (ntp) to transmit international atomic time (tai). In *32nd Annual Precise Time and Time Interval Systems and Applications Meeting; 28-30*, pages 431–440, Reston, VA; USA;, 2000.

[10] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[11] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE TDSC*, 1(1):Page(s): 11–33, 2004.

[12] BIMP, IEC, IFCC, ISO, IUPAC, and OIML. *ISO International Vocabulary of Basic and General Terms in Metrology (VIM)*, third edition edition, 2004.

[13] Peter H. Dana. Global Positioning System (GPS) time dissemination for real-time applications. *Real-Time Systems*, 12(1):9–40, January 1997.

[14] HIDENETS. Deliverable 3.3 - mechanisms to provide strict dependability and real-time requirements, 2008.

[15] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev.*, 36(SI):147–163, 2002.

[16] A. Bondavalli, A. Ceccarelli, and L. Falai. Reliable and self-aware clock: complete description. Technical report, University of Florence, 2008.