Using simple diffusion to synchronize the clocks in a distributed system.

Augusto Ciuffoletti
Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa (Italy)

Abstract

Simple diffusion consists in propagating some data or computation to every unit in a network by spreading it from one unit to its neighbors, which in their turn spread it to the neighbors until every unit is reached. Simple diffusion is easy to implement, scalable, robust with respect to omission faults, and privileges local communication (i.e., uses only links whose cost is the lowest in terms of time needed and/or traffic generated).

A solution to the problem of keeping synchronized all the clocks in a large distributed system is presented, that is based on simple diffusion. The traffic generated is $O(N)$, where N is the number of units in the network. The attainable precision is $O(d * \varepsilon)$ where ε is the precision with which a unit can synchronize its clock with that of a neighbor, and d is the maximum distance between a unit and the nearest reference clock.

Key words: Distributed Computing, Clock Synchronization, Rumor Diffusion, Self-Stabilizing Algorithms, Synchronous Multicast, Local Synchronization.

1 Introduction

The control of distributed activities is greately simplified if all the nodes of a distributed system share a common clock value [11]. Unfortunately, it is impossible to implement a service that exactly matches this assumption: whenever a timing signal is transmitted from a site to another, no matters whether the two ends are CMOS gates in a chip or mainframe computers in a WAN, the signal is exposed to random phenomena that alter it, thus making imprecise the global timing service. If the timing error is bounded and sufficienly small, the design can take into account that source of non-determinism, and compensate the forecast timing imprecision by slowing down the activity: as an effect, the precision of the global timing service has a direct impact on the performance of the system.

If the global timing service is shared by a small number of units that are physically near, we can provide a unique device that measures the time, and that transmits the timing signal to all the units along a common communication medium (see Figure 1).

In many cases, the complexity of the system justifies the provision of an independent time measuring

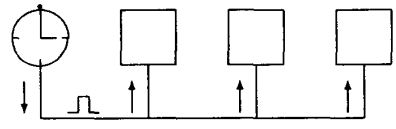

Figure 1: A bus based timing system

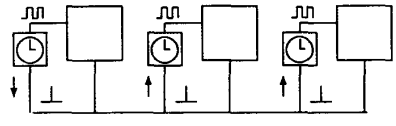

Figure 2: A bus based timing system with independent timing devices

device for every unit: a clock is a small and inexpensive device (see Figure 2). But independent time measures are affected by independent errors. To avoid cumulative effects on the measures of very long time intervals, from time to time the units must be resynchronized.

An advantage of using independent clocks periodically resynchronized is that, being the resynchronization a quite infrequent event, a dedicated line devoted to the implementation of the distributed timing service is not needed. A hardware implementation of this concept is in [10]: to avoid the loss of precision due to the bus contention delays and to the message handling performed by the software, the resynchronization information is placed on the transmitter only when the access to the bus is granted.

If the application requires a high degree of reliability, we can arrange to tolerate the failure of one or more time measuring devices: a survey of the work done on this subject is in [15], and the implementation referred above also ensures a high reliability. Ring architectures are also applicable to implement a bus based global timing system, as illustrated in [8].

But bus based approaches are of limited scope, since the timing imprecision is bound to the lengh of the bus and to the number of units attached to it, and by the intrinsic limitations of bus based architectures

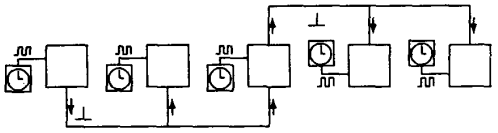

Figure 3: A timing service on a system supported by two buses

[9].

When the number of units in the system prevents the use of a unique bus to share the timing signal, a resort is to use intermediate units to relay the timing signal from one bus to the other.

Note that this perspective is not limited to medium or wide area networks (see [12] for an early work): MIMD systems that contain thousands of autonomous processors are a reality (e.g., CM5 by Thinking Machines Corporation).

If the timing signal is transmitted like an ordinary message with a datagram like protocol, the unpredictability of the message handling delays (induced by routing, input and output queues management, scheduling and other non-deterministic activities), would degrade the precision of the common timing service. In [5] the author presents an algorithm that maintains a clock of a unit (server) synchronized with a clock located on a remote unit (master): the quality of the results are bound to the shape of the probability distribution of the delays: more precisely, to the difference between the minimum and the average time required for a message to complete a *round trip*, from the server to the master and back.

In [5] the author claims that it is reasonable to assume that the round trip delay has a Gamma-like distribution, and reports experimental values for the minimum value, and the median , respectively around 4.2 and $4.5 msecs$. Under these and other reasonable assomptions, the attainable precision is about $1 msecs$.

We can identify two major drawbacks in a global time service based on datagram communications:

- much of the imprecision is due to the (unpredictable) message handling overhead, and is far from the physical limits;
- intermediate units cannot use the timing messages they relay to support the local instance of global clock service, since the datagram protocol does not guarantee that two messages respectively from unit a to b and back follow the same path. Each unit must use distinct message streams to implement the service.

An alternative way consists in exploiting bus communications to implement a precise and cheap common clock service for those (small) sets of units that share a bus. In the less favorable case represented by a mesh system, each bus connects only two units. The timing service is then propagated from set to set: those units that are directly connected to the time measurement device will implement a local time service, and transmit the timing signals to the other sets they belong to. One major problem of this approach is the need of avoiding *loops* in the synchronization timing transmission subnet: to this purpose, the system is organized in a hierarchy.

The Internet timing protocol NTP [13] is implemented following this idea. The overall network is therefore organized in *strata*: at statum 1 clocks are extremely stable and communicate through fast and reliable media (as short and micro waves broadcasts). At lower strata the clocks are conventional crystal oscillators and communicate through telephone lines. A unit always chooses a unit in a higher strata to obtain the timing information.

This paper illustrates an algorithm that is based on the above idea but leaves the hierarchy quite dynamic. The reference architecture can be represented by a hypergraph: each hyperarc connects a unit to a set of other units to which it can directly send a timing signal, and from which it can receive a request for a timing signal. The timing signal consists of of information that allows the receiver to resynchronize a local time measuring device within a given distance from the time measuring device of the source of the signal. The receiver will then use the (now rather precise) local time measure to synchronize other units connected to it but not to the original timer. We call this technique *simple clock diffusion*.

Although this basic idea induces a sort of a hierarchy, this is in fact dynamic and depends on how fast the timing information is diffused: absence of loops is implied by a three state protocol. In addition and unlike NTP, all units and communication links share the same delay characteristics: in other words, we assume that the network is homogeneous from the point of view of clock diffusion.

We envision the use of this kind of algorithms within a multicomputer composed of hundreds or thousands of units: typical applications of such systems include discrete event simulation. Since the protocol does not explicitly introduce a binding between the real time and the values of the clocks that are kept synchronized, it could be used also to maintain a uniform virtual timing in the system.

An early example of clock synchronization algorithm based on simple clock diffusion is in [4]: the authors assume that the system is absolutely democratic, since none of the participants has a privileged clock. Each time the synchronization algorithm is run, an implicit form of elections occurs to agree a clock value. As a consequence, the system clock will run as fast as the fastest clock in the system, with no possibility of synchronizing the system with respect to an external reference. The authors indicate how a joining processor can get synchronized with the rest of the system, but do not say how the system behaves at start-up.

This introduces a qualifying characteristic of the protocol presented in this paper: in fact, it is possible to prove that it is *self-stabilizing*, in the sense that it does not need a bootstrapping procedure that brings the system into a legal state, and that the units never need to know the global state in order to perform con-

sistently. As a side effect, the protocol survives silent failures and provides the units of a way of dynamically joining the system, but needs the presence of some privileged units that keep a consistent view of the same timing reference.

We will not refer to two widely accepted concepts concerning clock synchronization: namely, the distinction between internal and external synchronization [10], and between virtual and physical clocks. As regards the first point, the protocol we introduce is a hybrid of internal and external synchronization: like in an external synchronization algorithm, we introduce the existence of a system wide time reference, but this is implemented by a pool of units that agree about an internal timing. Whether this internal reference be or not bound to real time is a question that does not affect the algorithm. As regards the concept of a virtual clock, as distinguished from the physical clock that renders the physical device, we do not need one: thereafter, we refer to a clock as a variable that is kept updated by a device that ensures a limited agreement with all the other clocks in the system, and that can be adjusted by the unit when needed. We say that it is used to measure the passage of time, but whether this is an internal or external time we do not say. Therefore, the clock as defined in this paper takes something from the virtual clock, since it can be adjusted, and something from the physical clock, since we assume that its value varies in accordance with a paradigm that is approximately agreed on by all the clocks in the system, like the period of the oscillation generated by a quartz crystal of given characteristics.

2 System assumptions

We assume that the network is composed of n units, and that a link is a bidirectional connection between two units. We admit link failures that may cause the loss of messages (*omission faults*), and unit failures that consist in the inactivity of the unit (*silent failure*). We assume the existence of other failure assumptions that specify what combination of failures is admissible, and that the system never partitions into several disconnected components as a consequence of admissible failures; in addition, we indicate the number of links in the shortest path connecting two units p and q even in the worst admissible case with $dist(p, q)$, i.e. the distance between two units,.

Each unit has access to two basic services:

synchronous multicast: consists of sending a message to every neighbor. We assume that this operation succeeds with high probability within a given time limit. Otherwise the information will be received outside the limits or not at all.

local synchronization: consists of synchronizing all the clocks of the correct neighbors with the clock of the unit that provides the service. Also in this case the operation usually succeeds within a given time limit, but we exclude the case of an inconsistent operation.

Both services can be implemented in a number of ways: first we give their formal specification, and then we outline how to implement them.

Definition 1 (Synchronous multicast) *Let p be a unit and $N_p = \{q_i, 1 \leq i \leq n\}$ be the set of p's neighbors. If p multicasts a message to N_p at time t_0 (i.e., when the clock of a reference is t_0), there exists a finite delay Δ_{com} (as measured by the clock of a reference unit) such that at time $t_0 + \Delta_{com}$ the message has very probably been received by every unit in N_p.*

The implementation of this service depends on the way the communication is supported in the system. If the network is based on direct links between pairs of nodes, a timed command will send a separate copy of the message to each of the neighbors; if the system has a broadcast facility to communicate with the neighbors (such as an Ethernet, or a parallel bus) this will be used to diffuse the message.

As regards the timing of the system, every unit has a clock, defined as a variable whose value should correspond to a globally agreed quantity that we call time; we associate to the clock another quantity that represents a pessimistic estimate of the accuracy of the clock. The value of the clock is updated by a timing device that can measure the passage of time with a given precision (e.g., a 1 Mhz quartz oscillator increments the clock every microsecond $\pm$ 1 picosecond) and the (in)accuracy also increases as time passes. The unit can either read the clock (this operation returns to the unit the present value of the clock and of its accuracy), or adjust the clock (this operation replaces the old value of the clock and of the accuracy with the new ones). In a timed system, the clock must comply to several requirements: we divide the definition of a properly timed system into two parts, and see for each of them why it is a reasonable assumption. In the next section we shall indicate the algorithm that attains a clock synchronization after these premises have been satisfied.

Definition 2 (Correctly timed system) *In a correctly timed system, there exists a set of units $\mathcal{R}$ such that,*

- *when an event a occurs, for any two units r, s in $\mathcal{R}$ we have*

$$|t_r^a - t_s^a| \leq \varepsilon_{ref}$$

for a given integer ε_{ref};

- *given a unit r in $\mathcal{R}$, and a unit p that does not adjust its clocks in the time between two events a and b, it holds that*

$$(1 - \rho) \leq \frac{t_p^b - t_p^a}{t_r^b - t_r^a} \leq (1 + \rho)$$

for a given real ρ;

- *an upper bound d is given for the distance between a unit p and the nearest unit q that is in $\mathcal{R}$;*
- *each unit can execute a local procedure $AmIRef$ that returns true if the unit is in $\mathcal{R}$, false if not.*

In essence, we assume the presence of a set of units that hold clocks that evaluate the present time in approximately the same way. The values of the clocks of the units in $\mathcal{R}$ (here after called reference units) fall within an interval spanning ε_{ref}: this is the property usually associated with a set of accurate clocks [15]. In addition, the bound drift property always holds for a reference unit and the units know whether they are members of the set of references or not: we assume that the set of references is statically predetermined, and therefore the answer to the $AmIRef$ procedure is hardwired into each unit.

This property is needed to establish the timing of the system: we introduce a privileged set of units, and an invariant property of the clocks of these units. Thereafter, when we mention the time at which a certain event occurs, we implicitly indicate its measurement on the clock of a properly working reference unit. The multiplicity of accurate clocks guarantees against failures: if a reference unit fails (by assumption it fails silently), the others can still maintain the timing of the system. In addition, if the references are uniformly distributed in the network, the synchronization is more efficient. To keep the references synchronized, we indicate the adoption of a specific hardware support: either a specific link among the references or the synchronization of each of them to an external real time source (UTC, GPS): if the time corresponds to a quantity other than real time, they can use any other way of measuring its variation, provided that this measure respects the definition given above. An alternative solution, but expensive, is by one of the known clock synchronization protocols [15], improved with a probabilistic clock reading rule [5]. Here we do not indicate a solution to this problem since it is an extension of other existing results.

The next step consists in defining a basic primitive to communicate the value of the clock among the units.

Definition 3 (Local synchronization primitive) *Let p be a unit and $N_p = \{q_i, 1 \leq i \leq n\}$ be the set of p's neighbors. t_p indicates p's clock when it starts the synchronization protocol:*

1. *two finite values Δ_{sync} and ε_{sync} are given such that when the value of p's clock is $t_p + \Delta_{sync}$ the value of the clock of any unit in N_p is adjusted within the interval $[t_p + \Delta_{sync} - \varepsilon_{sync}, t_p + \Delta_{sync} + \varepsilon_{sync}]$;*
2. *each of the neighbors obtains an upper bound ε of the synchronization error, therefore when the value of p's clock is $t_p + \Delta_{sync}$, any unit in N_p knows that the value of its clock is within the interval $[t_r - \varepsilon, t_r + \varepsilon]$ for any reference r.*

We do not address the specific implementation of this primitive. It poses the usual dilemma: hardware or software? Several clocks can be synchronized by using dedicated hardware devices and an appropriate communication protocol [10] or by using the message exchange support offered by the distributed operating system [3](in our case, the synchronous multicast primitive could serve this purpose). Here we prefer to leave the question open. Note that, during this operation, the sender acts as a master and all the neighbors *reach contact* with it (using the terminology of [5]) if no failure occurs. When a unit gets synchronized to some other unit, it also obtains a pessimistic approximation of the synchronization error.

3 Synchronization requirements

We require the protocol to guarantee a given degree of accuracy for every clock in the system. In addition, the system overhead imposed by the clock synchronization service must be reasonable.

Definition 4 (Clock synchronization service) *The clock synchronization service must satisfy the following specifications:*

quality *when an event a occurs, for any two units p, q of which one is a reference we have*

$$|t_p^a - t_q^a| < \varepsilon_{max}$$

with $\varepsilon_{max} \in O(d * \varepsilon_{sync})$

cost *during a time interval of $\frac{\varepsilon_{max}}{\rho}$ time units, the clock synchronization service requires the execution of $O(N)$ synchronization or multicast primitives.*

The quality requirement indicates that the accuracy should be bound to the maximum distance between a unit and the nearest reference, and to the precision with which a clock can synchronize its neighbors by using the local synchronization primitive. The cost requirement indicates that the algorithm must be optimal, considering the O of the number of synchronizations and message exchanges performed. In fact, each unit must be resynchronized at least every $\frac{\varepsilon_{max}}{\rho}$ time units, and since resynchronizing entails reading another clock, during that lapse certainly $O(N)$ synchronizations have to be performed, even in the extremely favourable hypotheses that each unit were a neighbor to a reference.

Note that the clock controlled by the algorithm defined above may exhibit discrete and non monotonic adjustments. Since this behavior is incompatibile with real applications, we should amortize the adjustments. To implement this we refer to [14], where the authors show that it is possible to amortize discrete, non-monotonic adjustments without losing precision.

4 Description of the algorithm

We introduce a simple event-driven description of the synchronization algorithm that highlights the cyclic nature of the algorithm.

Each unit is described as a finite state automata. The set of states is made up of three elements: *steady*, *hungry*, *reference*. A reference unit is always in the *reference* state. All non-reference units move from one state to the other in accordance with an algorithm which is identical for every unit. The change of state is triggered either by local time-outs, or by interaction with the neighbors. The latter case consists in the receipt of a piece of information sent by one of the neighbors. In our case, we restrict to two possible kinds of interaction:

- one which is produced when a unit invokes the *synchronousmulticast* primitive to send the neighbors a request of synchronization. We label this event *synchrorequest*;
- one which is produced when a unit invokes the *localsynchronization* primitive to offer the neighbors the opportunity to synchronize their clocks with its own. We label this event *synchrooffer*.

Note that the above events have a fan-out greater than one, and, by the definition given in the previous section, have a non-negligible but limited duration. In addition, the state transition can be triggered by a timeout set on the local clock. A timeout is set by a unit p which produces the event *timeout*⟨*name*⟩*at*⟨*expr*⟩ that is invisible to other units, and the activation of the timeout is represented by the event ⟨*name*⟩*expired* that is only observed by unit p. A timeout can also be cancelled by producing the event *revoke*⟨*name*⟩.

Let us analyze the transition graph that represents the algorithm, beginning with the *reference* state.

A unit remains in this state as long as its clock is estimated to be sufficiently close to the time of the reference units to be used to synchronize other units: ε_{steady} indicates the maximum deviation from the time indicated by a reference unit which is allowed to a unit in the *reference* state. By necessity $\varepsilon_{steady} > \varepsilon_{ref}$. When in this state, a unit acts as a reference unit: i.e., it immediately responds with a *synchrooffer* to any *synchrorequest*. As we will see, this happens tipically in response to an exceptional event, like the restart of a part of the system; a unit leaves the *reference* state as soon as it estimates that the inaccuracy of its clock exceeds ε_{steady}, and moves to the *steady* state.

A unit that is in the *steady* state has a clock value which is still within the required limits (ε_{max} from a reference clock), but is not sufficiently precise to be used as a reference to synchronize other units; it moves to the *hungry* state as soon as it needs a preciser value for its clock. This happens when one of the following events occurs:

- the current clock error will become unacceptable in a time that is nearly the time required to get resynchronized;
- one of the neighbors requests a clock resynchronization (*synchrorequest*).

Note that the first event occurs after a fixed time τ_{steady} after the unit enters the *steady* state: this lapse corresponds to the time necessary to attain the maximum allowed inaccuracy under the maximum possible drift, minus the maximum lapse between the production of a request and the observation of an offer.

As soon as one of the above events occurs, the unit produces a *synchrorequest* and enters the *hungry* state: it will match and ignore further *synchrorequest* events, waiting for a *synchrooffer*. If an observed *synchrooffer* is not sufficiently precise (an infrequent event) the unit remains in the *hungry* state.

The unit leaves the *hungry* state as soon as it matches a *synchrooffer* and enters again the *reference* state producing in its turn a *synchrooffer*.

The above informal description is represented in Figura 4: each state is labeled with its name, and each arrow with a pair consisting of an event that triggers the transition, and an action that is performed by the unit during the transition. For typographical reasons circular arrows are not represented.

Let us see what happens when a unit p moves to the *hungry* state, and produces a *synchrorequest* event. After less than Δ_{com} time units, every neighbor in the *steady* state will observe the event and move to the *hungry* state in its turn, and will produce another *synchrorequest*. In essence, p generates a *synchrorequest* wave that propagates to every unit in the *steady* state which is a neighbor to a unit in the wavefront. The *hungry* units that the wavefront encounters are part of a similar wavefront, and the two will merge. In other words, the wavefront encloses a connected component of the graph which only contains *hungry* units. The wavefront freely propagates until it encounters a unit in the *reference* state. This unit will respond to the *synchrorequest* by producing in its turn a *synchrooffer*, and remaining in the *reference* state. The neighbor units (and, among them, the one which produced the *synchrorequest*) will observe the *synchrooffer* and will move to the *reference* state which then produces a *synchrooffer* event. Other neighbor units will then move to the *reference* state, giving rise to a reflected wave that will move backward. If the precision of the clock originating the reflected synchronization wave is enough, p will eventually be synchronized: but we have to take into accout the possibility that the synchronization wave dampens before reaching p. The dampening of the synchronization wave is due to the fact that each time a clock value is diffused one step further on the network, it looses some precision, as stated in the definition of the local synchronization primitive.

In the case the wave dampens, either p is reached by another synchronization wave with a different origin, or it will have to wait until all the units in the reference state in the dampened wave have time to move spontaneously to the steady state, leaving the way to the former request wave which will (hopefully) trigger a sufficiently precise synchro wave. Note that such units cannot detect that they are part of a preco-

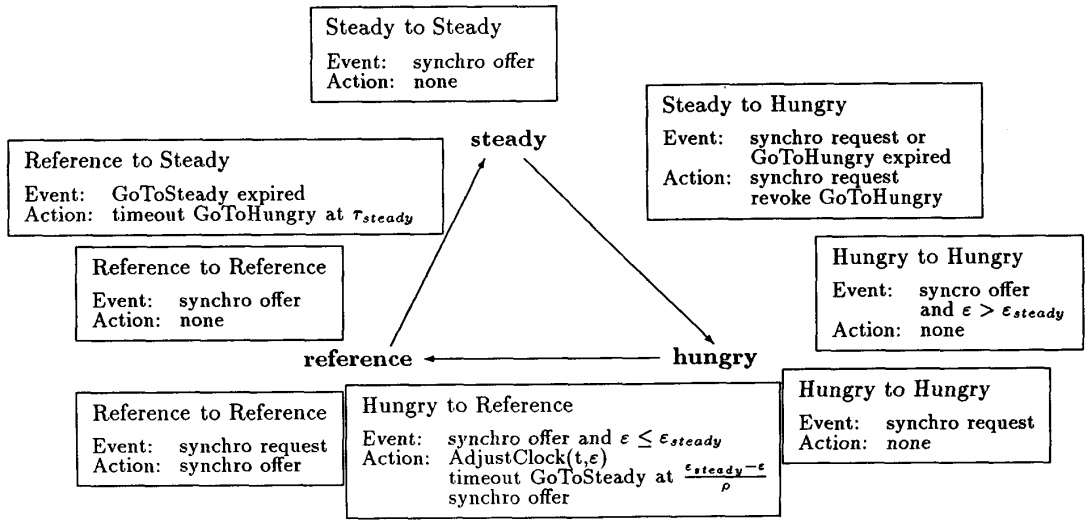

Figure 4: State transition graph of the synchronization algorithm

ciously dampened synchronization wave: neither the precision of the local clock, nor the negative acknowledgement coming from the partner that failed to synchronize can univoquely indicate this event.

As a consequence, admitting the possibility of synchronization waves that dampen too soon makes the time between the generation of a synchronizaton request and its fulfillment hardly predictable, and this reflects negatively on the cost of the algorithm. We shall keep into account this indication in the sizing of the values that determine the work of the algorithm.

4.1 Maximum attainable precision and other timing issues.

This section indicates the way to tailor the algorithm to a specific situation, once the constant values defined in the system model section have been determined. We therefore define the admissible values for the parameters ε_{steady}, ε_{max}, and also give some informal indications of the optimal choice for these parameters.

The first step in our discussion consists in describing the desired stable properties that the system state must respect, and that the algorithm must implement:

- the clock *quality* requirements are fulfilled by every unit and
- when a component enters the hungry state, as a general rule any other non-reference unit is either in the steady or in the hungry state.

The former requirement needs no words, the latter translates the indication with which we concluded the last section.

Let us draw some conclusions about the values of the time variables that characterize the algorithm: in order to fulfil the first requirement, a unit enters the hungry state and a request wave is be generated before the local error attains the value ε_{max}. In order to give to the request wave enough time to reach the reference, and to the synchronization wave th time to reach the origin of the request wave, the following inequality must hold:

$$\varepsilon_{hungry} \leq \varepsilon_{max} - \rho * d * (\Delta_{com} + \Delta_{sync}) \quad (1)$$

In addition, to guarantee that a synchronization wave generated by a reference unit reaches the components at distance d, we have to ensure that such a propagation accumulates an error that is lower than the threshold ε_{steady}, that regulates the acceptability of a the local synchronization:

$$\varepsilon_{steady} \geq d * (\rho\Delta_{sync} + \varepsilon_{sync}) + \varepsilon_{ref} \quad (2)$$

To ensure that no non-reference unit is in the reference state when another unit moves to the hungry state we must be able to ensure that all units that entered the reference state during the previous synchronization wave leave this state before the next request wave.

Let us take t_0 as the value of the clock of the reference when it produces the synchronization offer, thus generating the synchronization wave when all units are *hungry*: the earliest unit to enter the *hungry* state is the one that receives instantaneously the synchronization offer (this is possible since we do not pose a lower bound to Δ_{sync}) and obtains the worst possible approximation so that it immediately leaves the *reference* state and enters the *steady* state: it will enter the *hungry* state when its clock has accumulated

enough error to bring it to the threshold ε_{hungry} i.e. at time

$$t_1 = t_0 + \frac{\varepsilon_{hungry} - \varepsilon_{steady}}{\rho} \quad (3)$$

On the other hand, the last unit which leaves the *reference* state is the one that observes a very late *synchrooffer*, but obtains an exact synchronization, so that it remains for the maximum time in the *reference* state; for this unit we obtain the time t_2, when it leaves the *reference* state:

$$t_2 = t_0 + d * \Delta_{sync} + \frac{\varepsilon_{steady}}{\rho} \quad (4)$$

Since we want $t_2 \leq t_1$:

$$\varepsilon_{hungry} \geq \rho * d * \Delta_{synch} + 2 * \varepsilon_{steady} d * \Delta_{sync} \quad (5)$$

The three inequalities 1, 2, and 5 are to be satisfied if we want that the system be characterized by the stable properties outlined at the beginning of this section. In that case, the behavior of the whole system will be cyclic, similar to that defined by a single unit: starting from the state where all non-reference units are in the reference state, they will spontaneously move to the steady state. When the first unit moves to the hungry state, all other non-reference units are in the steady state, and will freely propagate the request wave. The successive synchronization wave will bring back the system to the state where all components are in the reference state.

We further need to prove that the system eventually falls in any of the three *unanimous* states, regardless from the initial state. The proof proceeds by steps, starting from a generic cahotic situation, characterized by random states and clocks associated to the units. We assume that reference units have clock values within the required limits.

From the initial situation, all units in the reference state but with arbitrary values for the clocks will spontaneously move to the steady state. From this time on the reference units will be free to exploit their activity. Next all units in the steady state but with arbitrary clock values will move to the hungry state. At this point, the system will contain units in the reference and in the steady state that have clock values and errors complying with the specifications of such states; there will be also a number of hungry units with either consistent or arbitrary clock values, values. They will generate a request wave that will eventually reach a reference, and finally a synchronization wave will bring all units in the reference state, with clock values complying with the specifications.

To conclude we derive a lower bound to the maximum precision attainable, which can be obtained by 1 and 5:

$$\varepsilon_{max} \geq \rho * d * (\Delta_{com} + 2 * \Delta_{sync}) + 2 * \varepsilon_{steady} \quad (6)$$

and, using 2

$$\varepsilon_{max} \geq \rho * d * (\Delta_{com} + 4 * \Delta_{sync}) + 2 * d * \varepsilon_{sync} + 2 * \varepsilon_{ref} \quad (7)$$

The dominating part of the sum is $2 * d * \varepsilon_{sync}$, since the first part of the sum contains a very small factor ρ, and we assume $\rho^{-1} \gg d$, and the last part does not contain a factor comparable with d. Therefore we can conclude that $\varepsilon_{max} \in O(d * \varepsilon_{sync})$, as required by the *quality* part of the definition of the clock synchronization service.

As regards the number of primitive calls used to perform a clock synchronization which brings each unit in the system into the *reference* state when every unit apart from the references is in the *steady* state, this operation clearly requires $n - n_{ref}$ (n_{ref} is the number of reference units) multicasts to propagate the request wave and bring every unit into the *hungry* state, and n *synchrooffers* produced by each *hungry* unit when receiving the *synchrooffer*. Therefore, as required by its specifications, the synchronization service requires $O(n)$ primitive calls every approximately $\frac{\varepsilon_{max}}{\rho}$ time units, and these calls are equally distributed among all the units of the system, provided that the collisions between request waves and poorly synchronized units in the *reference* state are avoided.

4.2 Robustness of the clock synchronization algorithm

The protocol presented in this paper is based on diffusing computations: this is one of the least critical ways of broadcasting, because the propagation eventually finds the best path to reach every reachable unit (even if some component fails), and because this search process is carried out in parallel. This entails two important properties: the predictabilty of local communication delays is amplified, and a moderately redundant network (such as a grid mesh) can tolerate arbitrary failure patterns.

The former property has been implicitely used in the computation of the maximum attainable precision: in fact, while the specification of the synchronous multicast has a probabilistic flavor ("...the message has very probably been received ...") this aspect disappears when this definition is used to compute a diffusion time (e.g. in refeq:smax). This is justified by the fact that diffusion process tends to improve the predictability of the single step, when it occurs in a large regular network: e.g. in a 8x8 grid torus (d=8) where a single propagation step, from a unit to a neighbor, takes more than one time unit with probability greater than $2 * 10^{-1}$, the entire diffusion process, from one unit to every unit, takes more than 8 time units with probability lower than $9 * 10^{-7}$. The proof of this simple result is quite complex: the interested reader can find an exhaustive proof in [2]. By means of this result, we can turn a strong assumption on each diffusion step into a weak assumption on the whole diffusion process.

As for the second property, it guarantees that the diffusion reaches all reachable components: however, the time needed to complete the diffusion is negatively affected by a failure, but the degradation is quite smooth, as shown in [7].

5 Conclusions

The distinguishing features of a solution based on simple diffusion are low cost, ease of implementation, expandibility and robustness. With respect with other solutions based on this elementary mechanism, namely [13] and [4], the solution presented in this paper is self stabilizing but relies on the existence of few privileged components in the system that keep the reference timing.

Self-stabilization is a concept that has been introduced by [6] and since then it is recognized as a desirable property of a distributed algorithm (see [1] for a survey). Clock synchronization is necessarily a self-stabilizing algorithm, and differs from other distributed computing issues since we cannot rely on the existence of a stable or final state of the system: the states of all units are continuously changing, although there is a certain regularity in the change. Moreover, the stability enforced by the execution of the synchronization algorithm is constantly destroyed by the drift of the clocks: it is vital that the algorithm is faster in stabilizing than the drift is in destabilizing it and, even when this basic property is satisfied, the occurrence of unpredicted events can jeopardize the regularity that guarantees stability. This paper accounts for an example of this issue.

Concerning the way we address the problem of clock synchronization, we have tried to focus on the unsolved problem, abstracting from those subproblems for which several solutions are known. Thus we have left unimplemented the local multicast and synchronization operations, providing the reader for an abstract and generalized definition of them. This definition has been exploited in the implementation of the global synchronization algorithm, which was our main objective. Similarly, we have limited our interest to a discrete clock synchronization service, since the implementation of a continuous clock synchronization service from a discrete one is a solved problem.

References

[1] Anagnostou E.; El-Yaniv R. More on the Power of Random Walks: Uniform Self-Stabilizing Randomized Algorithms. In: Distributed Algorithms - 5th International Workshop, WDAG '91; Eds: Toueg S.; Spirakis P.G.; Kirousis L.; Lecture Notes in Computer Science; 579: 31-51.

[2] Ciuffoletti A. "Reliability vs cost: design of a probabilistic broadcast algorithm", *Distributed Computing* (1994):8 (to appear).

[3] Ciuffoletti A., Gattai F., and Golinelli R. "Clock Synchronization in Virtual Rings", in Proc. Euromicro Workshop on Real-Time Systems, Vaesteraas, Sweden, June 1994. (in print)

[4] Cristian F., Aghili H., and Strong R. "Clock Synchronization in the presence of omission and performance Faults, and Processor Joins", in Proc. of the 16th FTCS, Vienna, 1986.

[5] Cristian F. "Probabilistic clock synchronization", *Distributed Computing* (1989):3:146-158.

[6] Dijkstra E.W. "Self-stabilizing systems in spite of distributed control" *Communications of the ACM* 1974; 17(11):643-644.

[7] Feige U.; Peleg D.; Raghavan P.; Upfal E. "Randomized Broadcast in Networks" In Proc. SIGAL Symposium on Algorithms, Tokyo; 1990. 1-19.

[8] Halpern J.Y.; Fagin R. "Modelling knowledge and action in distributed systems" *Distributed Computing*, (1989):3:159-177.

[9] Hartenstein R.W.; Koch G. "The Universal Bus Considered Harmful" in: R.W. Hartenstein and R. Zaks. *Microarchitecture of Computer Systems*, North-Holland Publishing Co, 1975, 119-130.

[10] Kopetz H.; Ochsenreiter W. "Clock Synchronization in Distributed Real Time Systems" *IEEE Transaction on Computers*, August 1987, 7(3):404-425.

[11] Lamport L. "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems" *ACM Transactions on Programming Languages and Systems*, April 1984, 6(2):254-280.

[12] Luetchford J.C.; Heynen J.; Bowick J.W. "Synchronization of a Digital Network" *IEEE Transactions on Communications*, August 1980, com-28(8):1285-1290.

[13] Mills D.L. "Network Time Protocol: Specification and Implementation. (version 1)" ed.: University of Delaware; July 1988; Report RFC-1059. (DARPA Network Working Group).

[14] Schmuck F.; Cristian F. "Continuous Clock Amortization Need Not Affect the Precision of a Clock Synchronization Algorithm" In Proc.: 9th ACM Symp. on Principles of Distributed Computing. 1990.

[15] Schneider F. B. "A Paradigm for Reliable Clock Synchronization", Technical Report of Cornell University Ithaca N.Y.; February 1986; TR 86-735.