Self-Stabilizing Diffusion

Augusto Ciuffoletti *
Dipartimento di Informatica - Universitá di Pisa
Corso Italia 40 - 56125 Pisa - ITALY
e.mail: augusto@di.unipi.it
url: http://www.di.unipi.it/ augusto/augusto.html

1 Overview

This paper deals with a peculiar instance of the consensus problem.

In our scenario, the system can be assimilated to an irregular network composed of clusters of processing units. Each unit holds a "copy" of a value, that changes continuously according with a known function. The basic problem we address is to keep all the copies sufficiently close to the real value, given that most units cannot keep the local copy of the value exactly updated, with a minimal computational load.

With respect to the "precision" of the local copy of the global value, we distinguish two kinds of units: a restricted number of units that continuously keep the correct value, upon which any other unit should agree, and the rest of the units, that can only estimate locally the current global value.

We do not investigate the way in which the former units keep their copies exactly updated: since we count a few units of this kind in the system, they may use an expensive strategy. On the contrary, we are interested in the behavior of the rest of the system.

The precision of the local estimate of the other units progressively degrades, and they need to periodically *refresh* their local estimate in order to keep it acceptably near to the global value. Our problem is the design of a protocol that performs this task.

The relevance of the problem should become evident if we recognize that the above description is a paraphrase of the clock synchronization problem: a few units in the system hold accurate (and expensive) clocks, while the others periodically synchronize their in-accurate (and cheap) clocks with the accurate ones. The interested reader can find a technical discussion of the problem and its practical solution in Internet in [7]: the protocol suite is known as Network Time Protocol (NTP).

*This work was partially supported by MURST Project "Algoritmi per Sistemi Paralleli e Distribuiti"

In NTP each unit follows a selfish behavior, reading the remote accurate clock when a given timeout elapses and synchronizing the local clock. This may cause a very high traffic load in a system composed of many units, since each clock reading operation uses several links from the unit to the clock synchronization server and back.

In this paper we propose a completely different strategy, and perform a first step towards its implementation.

The strategy basically consists in generating a *synchronization wave* in response to a request: therefore many units will be able to synchronize their clocks during the same wave diffusion. We can quantify the difference among the two approaches by counting the link operations required to synchronize all the units in the system composed of n units with $O(n)$ links: if the average distance (in terms of links to cross) of a unit from a time server is $O(\sqrt{n})$, the operation requires $O(n^{3/2})$ link operations following the NTP approach, and only $O(n)$ following the synchronization wave approach.

The concept of wave diffusion is typical of synchronous networks, like systolic arrays, but is seldom applied to a-synchronous networks, as in our case. An application of this concept to clock synchronization in an asynchronous network can be found in [4]: in that case, the author assumes that there is not an "absolutely correct" value, and the units may converge to an arbitrary value, but common to all units. With this paper, we share also a common *fault model*, which will be described in the next section.

The operation that has to be carried out cannot be assimilated with that usually referred as *gossiping* [6]: in our case, communication is *some-to-all*. It is also more complex than that usually referred as *broadcast*, since we consider the interactions among different broadcasts in the system. In fact, we consider both the diffusion of the request and the one of the broadcast, and we pay special attention to the stationary process of request-broadcast that determines

a periodic, systolic behavior of the system.

In [2] we proposed an algorithm following the above guidelines. The algorithm was described informally and tested in a simplified environment. In the present paper we make a first step towards the formal statement and proof of this algorithm. Therefore we reworked a description of the problem that disregards the topics related to real time, and we wrote and proved the correctness of an algorithm that solves this limited instance of the problem. The insigth gained in the informal part of the work gives us confidence in the possibility to extend this algorithm to the real-time related problem.

One of the interesting features of the algorithm is self-stabilization. The interest for this feature comes from the consideration that the algorithm should reside at a very low level; therefore we cannot rely on powerful fault tolerant features. Our fault hypotheses exclude the permanent failure of a node, but otherwise analyzes any other kind of faults. However, in response to severe faults, the response of the algorithm may be less safe: a system controlled by a self-stabilizing algorithm may run into an inconsistent state when units fail or are restarted in an inconsistent state, but its consistency will be restored if all units work properly.

In the rest of the paper we will make little reference to the original problem of clock synchronization. Instead, we will define the problem starting from scratch, and define and prove the validity of a solution using a temporal logic, that excludes any possibility of representing real time.

2 System Model

The system is composed of interconnected *units*. A distinguished subset of such units is made of the units that we call *TRU*s (Time Reference Units): they are the source of the information to be diffused. In the following, we will use non-ambiguously the word *item* for the piece of information diffused by a TRU. When the system is in an inconsistent state, it may occur that some units behave like TRUs although they are not: we call them *false TRU*s.

The communication network is represented by a neighborhood relation: to each unit we associate a set of nodes that we call *neighbors*. The relation of neighborhood is symmetric. A unit can receive the item coming from different TRUs. For each unit and each TRU α, there is a subset of the neighbors of the unit from which it accepts the item: we call such neighbors the *synchro-neighbors* with respect to TRU α. A TRU is always a synchro-neighbor with respect to itself for each of the neighbors. The relation of *synchro-neighborhood* is a-cyclic. The notation used in the rest of the paper is summarized in table 1.

$\mathcal{T}$	the set of TRUs
α,β,γ	generic TRUs
x, y, z	generic units, but not TRUs

Table 1: System notation

The above definitions confine the spread of an item into a number of subgraphs: each of them has a pre-ordering structure (since it is a-cyclic), and is "rooted" in a TRU. The reason for this restriction is related to the original problem: the timing information carried by the item degrades as it is transmitted from a node to another, due to the indeterminacy of the communication delay, thus we have to limit the length of the path covered by an item.

We require a further property to hold, which restricts the way different diffusion subgraphs intertwine:

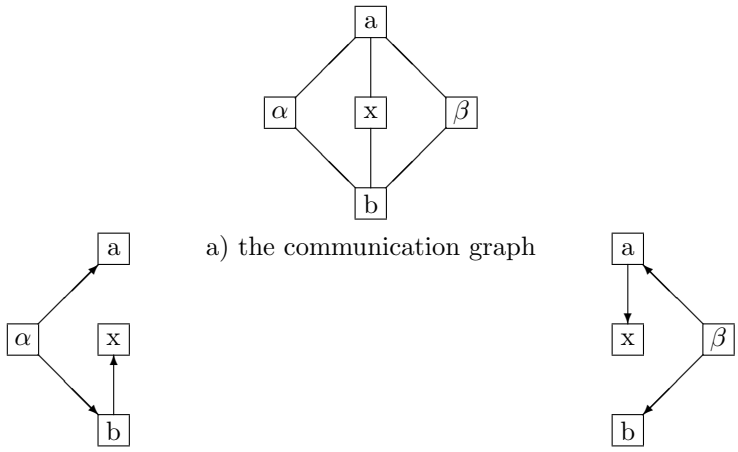
Predicate 1 *For all nodes x, there exists a neighbor y such that, if y has a synchro-neighbor with respect to the TRU α, then y is a synchro-neighbor to x for the TRU α.*

In other words, we require that each unit has at least one neighbor that can pass it the item, whatever the TRU is from which the neighbor has received the item. This condition is essential to avoid the case when a unit x never receives the item, since all synchro-neighbors repeatedly receive it from TRUs for which they are not synchro-neighbors to x. For instance, if the system is the one depicted in Figure 2a and the diffusion graphs are those depicted in Figures 2b and 2c, it might happen that unit x never gets the item: it happens when unit a privileges receiving the item from TRU α, and unit b privileges TRU β.

In the following, we will implicitly restrict any statement to the systems that satisfy the above Predicate 1.

Requirement 1 is quite restrictive, but it is satisfied by several topologies of interest: for instance, by any network where all units belong to only one synchronization graph, as well as every network where all units belong to all synchronization graphs. There are also regular topologies that satisfy Requirement 1: for instance, a square grid $n * n$, where TRUs are at the vertices of the network and, for each unit, the synchro-neighbors with respect to α are all the neighbors that are nearer to α, if the distance from the unit to α is less than n.

We adopt the fault model that is usually adopted for self-stabilizing algorithms: the system can reestablish a consistent state regardless of the state



a) the communication graph

b) the synchronization graph of TRU α

c) the synchronization graph of TRU β

Figure 1: A case of synchronization graphs that do not satisfy the Predicate 1

produced by the fault, once the fault has been removed. In addition, the disconnection of one or more units is not regarded as a fault, as long as the resulting subgraph satisfies the properties stated above. The behavior of the system while a fault is active, or during the time between its removal and the reestablishment of a consistent state, strongly depends on the type of fault. This topic, which is common to any self-stabilizing algorithm, will not be discussed in this paper, but we prove that the system will ultimately converge to a consistent state.

Another peculiar, but ultimately non-restrictive, hypotheses that we introduce is that the item generated by a TRU is unforgeable and unique. This is reasonable, since the granularity of the timing information carried by the item is so small, that the probability that a clock arbitrarily set up has the right value is negligible. But we argue (still have not proved) that even in this case the algorithm would not be disrupted.

3 The Algorithm

We assume that the state of each unit, except TRUs, is represented by:

- a label, that indicates the phase in which the unit resides with respect to the algorithm, and
- the name of a TRU (e.g., α), and
- an integer value (e.g., n).

The last two fields should uniquely identify (if the unit works properly) the item which was most recently received by a TRU: they indicate the n-th item generated by unit α. Since the items generated by a TRU are unforgeable and unique, if the two values above are correct they indicate a unique item.

The *label* can have one of three (mutually exclusive) values:

w(aiting) when the unit is waiting for a new item from a TRU;

s(teady) when the unit is satisfied of the item it currently holds;

r(eference) when the unit holds the most recent item diffused by a TRU, and can therefore diffuse it to the neighbors that are waiting for it.

The state of a TRU is uniquely identified by its name and by the number of the last item it released.

In Table 2 we summarize the notation used to indicate the local state of a unit.

We want that the algorithm exhibits three fundamental properties, that we informally state as follows:

progress: a waiting unit will eventually receive an item, and

self-stabilization: if all units participating in the algorithm are properly working, then the system will converge into a state where all units in the state r hold the most recent item diffused by the TRU, and such property will be satisfied as long as the units work properly. This property characterizes a *legitimate* state.

In order to realistically represent a clock synchronization algorithm, we also need a further property to hold:

$\langle lsp \rangle$	w_α	the unit is in the w state, and is in the synchronization graph of TRU α
	s^n_α	the unit is in the s state, and received the n-th data produced by TRU α
	r^n_α	the unit is in the r state, and received the n-th data produced by TRU α
	R^n_α	is the TRU α that holds the n-th data it produced

Table 2: The notation for the predicates related to the state of a unit

$\langle nsp \rangle$	$\overline{\langle lsp \rangle}$	a neighbor exists whose state satisfies $\langle lsp \rangle$ and it is a synchro-neighbor
	$\langle lsp \rangle$	a neighbor exists that is in state $\langle lsp \rangle$
	$\underline{\langle lsp \rangle}$	a neighbor exists whose state satisfies $\langle lsp \rangle$ and the unit is a synchro-neighbor for it

Table 3: The notation for the predicates related to the state of the neighbors of a unit

liveness: each unit perpetually loops through the three states.

This requirement is implicit in the definition of self-stabilization given by Dijkstra [5], but recent interpretations of self-stabilization consider it as too restrictive [8]. Note that the *progress* property stated above is a straightforward consequence of the *liveness property*.

Since the algorithm depends not only on the local state of the unit, but also on the state of neighbors, in Table 3 we give a concise notation also for the state of the neighbors of a given unit.

Based on such syntax, a predicate concerning the state of a unit as well as that of its neighbors will be indicated as:

$$\langle lsp \rangle [\langle nsp \rangle_1, ..., \langle nsp \rangle_n]$$

The global algorithm is composed of all the algorithms which are run locally by the units. Following the terminology introduced by "Unity" (see [1]) we say that they are composed by *union*. We have different algorithms for TRUs and for the other units: they are illustrated in Table 5 and 4

The self-stabilizing predicate, that is characteristic of a legitimate state of the system, can now be formulated as follows:

Predicate 2 (Legitimate state) *For every unit x in the r state, there exists a TRU α such that*

state $= r^n_\alpha$	if	$w[\overline{r^n_\alpha}, \neg \underline{s_\alpha}]$ $\square$
state $= s$	if	$r_\alpha[\overline{r_\alpha} \vee \overline{R_\alpha}, \neg \underline{r_\alpha}, \neg(w_\alpha[\neg \underline{s_\alpha}])]$ $\square$
state $= s$	if	$r_\alpha[\neg \overline{r_\alpha} \wedge \neg \overline{R_\alpha}]$ $\square$
state $= w$	if	$s_\alpha[\neg \overline{r_\alpha}]$

Table 4: The algorithm run by a non-TRU

state $= R^{n+1}_\alpha$	if	$R^n_\alpha[\neg \underline{r_\alpha}, \neg(w_\alpha[\neg \underline{s_\alpha}])]$

Table 5: The algorithm run by a TRU

$$r^n_\alpha \wedge R^n_\alpha$$

In order to prove the self-stabilization of the algorithm, we need a *metric*, a quantity that represents the *distance* of the present state from one of the legitimate states. As one may expect, this quantity is strongly related to the number of units that do not satisfy the self-stabilizing predicate.

We call *false TRUs* those units that do not respect the above property: namely those that have apparently been the source of an item, but are not TRUs.

Definition 1 (The metric) *We call* false TRUs *the units that do not satisfy the* legitimate state *predicate above, and for which the predicate $r^n_\alpha[\neg \overline{r^n_\alpha} \wedge \neg \overline{R^n_\alpha}]$ is valid. Let $\mathcal{F}_x$ be an integer value corresponding to the number of units that follow a false TRU x in the synchronization tree associated with α. Our metric $\mathcal{F}$ is the sum of the $\mathcal{F}_x$ over all the false TRUs in the system.*

Note that $\mathcal{F}$ may be higher than the number of units in the system. In addition, it necessarily decreases each time the third statement of the algorithm is used (see Table 4).

The fact $\mathcal{F} = 0$ is a sufficient and necessary condition for the absence of false TRUs in the system, and therefore can be used as an indication of a legitimate state.

The next step is to prove that self-stabilization and liveness hold for a system running the algorithm above. The proof will be carried out using Unity.

4 Outline of the Proof

The complete proof of the lemmas introduced in this section have been omitted from the paper published in the *ERSADS '97* proceedings. The interested reader is invited to download or request the complete paper [3] that contains the formal proofs.

The first statement to be proved is that the metric $\mathcal{F}$ is eventually decreasing.

$$(\mathcal{F} = k_{\mathcal{F}}) \mapsto (\mathcal{F} < k_{\mathcal{F}}) \quad (1)$$

This immediately leads to the proof of self-stabilization.

The proof that $\mathcal{F}$ does not increase entails proving that a unit that has no false TRUs among its ancestors in any of the synchronization graphs of which it is part, cannot receive an item from a false TRU. This is, indeed, evident.

The proof that $\mathcal{F}$ eventually decreases is more complex: the following Lemma plays a key role in the proof.

Lemma 1 (Self-stabilization property) *For each properly working unit,*

$$(r_\alpha \wedge (\mathcal{F} = k_{\mathcal{F}})) \mapsto (s_\alpha \vee (\mathcal{F} < k_{\mathcal{F}}))$$

If we assume that the current state is not legitimate, then it holds that $\mathcal{F} > 0$, and there is at least one unit which is a false TRU. When we apply the lemma to this unit, we conclude that the system will necessarily evolve into a state where

- the unit has moved into the s state, thus leaving the state of false TRU, or
- the metric $\mathcal{F}$ has decreased.

But, if the unit has moved into the s state, then it will not contribute to $\mathcal{F}$ any more, since it is no more a false TRU, and therefore the value of $\mathcal{F}$ decreases also in this case.

Thus we conclude that Lemma 1 is true, and that false TRUs will ultimately disappear.

The following lemmas hold when we have ensured the absence of false TRUs:

Lemma 2 (Liveness property) *For each properly working unit, when* $\mathcal{F} = 0$*,*

$$r_\alpha \mapsto s_\alpha \quad (2)$$
$$s \mapsto w \quad (3)$$
$$w \mapsto r_\alpha^n[\overline{r_\alpha^n}] \quad (4)$$

The relation 4) is the progress property for units in the w state, and the relation 2) is proved as a corollary of Lemma 1. Together they prove that units perpetually move through the three states (the *liveness* property stated before).

References

[1] K.Mani Chandy and Jayadev Misra. *Parallel Program Design.* Addison Wesley, 1988.

[2] Augusto Ciuffoletti. Using simple diffusion to synchronize the clocks in a distributed system. In *Proc. of the 14th IEEE International Conference on Distributed Computing Systems*, pages 484–491, Poznan (Poland), 1994.

[3] Augusto Ciuffoletti. Self-stabilizing diffusion. Available at ftp://ftp.di.unipi.it/pub/Papers/ciuffoletti/selection/ERSADS97.ps, 1997.

[4] Flaviu Cristian. Clock synchronization in the presence of omission and performance faults, and processor joins. In *Proc. of the 16th IEEE Fault Tolerant Computing Symposium*, pages 218–223, Vienna (Austria), 1986.

[5] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[6] Sandra M. Hedetniemi, Stephen T. Hedetniemi, and Liestman Arthur L. A survey of gossiping and broadcasting in communication networks. *Networks*, 18:319–349, 1988.

[7] Dave L. Mills. Internet time synchronization: the network time protocol. *IEEE Trans on Communications*, COM-39(10):1482–1493, 1991.

[8] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 26(1):45–67, 1993.