STABILIZING UNISON

Mohamed G. Gouda and Ted Herman

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-90-01 January 1990

Abstract

We present an elegant implementation of "clocks" in distributed synchronous systems. The implementation is stabilizing in the following sense. Starting from any state, the clocks are guaranteed to reach "unison" where they show the same time, and the shown time is incremented in each step.

Keywords: Clock synchronization, realtime, self-stabilization, synchronous systems, unison.

1 Problem

Consider a finite, undirected, and connected *graph*. Associated with every node i in the graph is a *variable* $x.i$, whose value ranges over the natural numbers, and a *program* $P.i$ of the form:

begin <action> [] ... [] <action> **end**

Each action is of the form

<guard> $\rightarrow$ <assignment statement>

The guard of an action in program $P.i$ is a boolean expression over the x variables and the local variables of $P.i$. The assignment statement of an action in $P.i$ updates variable $x.i$ and the local variables of $P.i$.

A *state* of this system is defined by a value for every $x.i$ and a value for every local variable in each program. An action whose guard is true at some state is said to be *enabled* at that state. Similarly, a program that has at least one enabled action at some state is said to be *enabled* at that state. We assume *maximal parallelism semantics*: starting from any state, the assignment statement of one enabled action in each enabled program is executed yielding the next state. Note that executing the assignment statements of different programs in parallel does not cause any conflict; this is because the sets of variables updated by these statements are mutually exclusive.

It is required to design the $P.i$ programs so that the following two properties are satisfied.

i. *Stabilization* [1]: Starting from any state, the system is guaranteed to reach a *unison state*; i.e., one where all the x variables have equal values.

ii. *Unison* [2]: Starting from any unison state, the value of every x is incremented by 1 in every step. (This implies that once unison is reached, it is maintained.)

2 Solution

Our solution requires that each program $P.i$ have a local variable v whose value ranges over the neighbors of node i in the graph. (Two nodes in the graph are called *neighbors* iff there is an edge between them.) Program $P.i$ can now be defined as follows.

```
begin     $x.i < x.v$  $\rightarrow$  $x.i, v := x.v,$        $neighbor.i$
       [] $x.i = x.v$  $\rightarrow$  $x.i, v := (x.i) + 1,$  $neighbor.i$
end
```

where *neighbor.i* is a procedure that returns an arbitrary neighbor of node i. We assume that if *neighbor.i* is invoked infinitely often, then each neighbor of node i is returned infinitely often.

3 Proof of Correctness

We show in this section that our solution satisfies the two properties of stabilization and unison.

Theorem 1: (*Stabilization*) Starting from any state, the system is guaranteed to reach a unison state.

Proof: Define a function *Rank* that assigns to each system state S a natural number *Rank.S* as follows.

$$\mathit{Rank}.S = n \times (\mathit{Range}.S) + \mathit{Top}.S$$

where	n	$=$	the number of nodes in the graph,
	$\mathit{Range}.S$	$=$	(the value of the largest x at state S)
			$-$(the value of the smallest x at state S),
	$\mathit{Top}.S$	$=$	the number of x's with the largest value at S.

Consider the following three propositions about function *Rank*.

i. For each state S, Rank.S $\geq n$

ii. If a state S is followed by a state S', then
$\mathit{Rank}.S \geq \mathit{Rank}.S'$, and
$(\mathit{Rank}.S = \mathit{Rank}.S') \Rightarrow$ $(\mathit{Range}.S = \mathit{Range}.S') \wedge (\mathit{Top}.S = \mathit{Top}.S')$ $\wedge$
$(\forall$ node $i : x.i$ is largest at $S \equiv x.i$ is largest at $S')$

iii. Starting from any non-unison state, function *Rank* eventually decreases.

Stabilization of our system follows directly from propositions i and iii; proposition ii is needed in proving iii. Proofs for these three propositions follow.

Proof of i: The value of *Top.S* is at most n and at least 1. Therefore, the value of *Rank.S* is smallest when $\mathit{Range}.S = 0$. At such an S, all x's have equal values which implies $\mathit{Top}.S = n$ and $\mathit{Rank}.S = n$. Thus, $\mathit{Rank}.S \geq n$ for every state S.

Proof of ii: In each transition from a state S to a next state S', every x whose value is smallest at S is incremented by at least one, every x whose value is largest at S is incremented by at most one, and each of the remaining x's either assumes the value of another x or is incremented by at most one. Thus,

$$\mathit{Range}.S \geq \mathit{Range}.S' \tag{1}$$

If $\mathit{Range}.S = \mathit{Range}.S'$, then at least one of the x's whose value is largest at S has been incremented in the transition from S to S'. In this case, every x whose value is largest at S', also has the largest value at S. Hence, we get:

$$(Range.S = Range.S') \Rightarrow (Top.S \geq Top.S') \tag{2}$$

From (1) and (2), and from the definition of *Rank*, we conclude

$$Rank.S \geq Rank.S' \tag{3}$$

The value of *Top* is at least one and at most n. Thus, the difference in the value of *Range* from S to S' cannot be compensated by a corresponding difference in the value of *Top* to keep *Rank* constant. Thus,

$$(Rank.S = Rank.S') \Rightarrow (Range.S = Range.S') \tag{4}$$

From (4) and from the definition of *Rank*, we conclude

$$(Rank.S = Rank.S') \Rightarrow (Top.S = Top.S') \tag{5}$$

If $Range.S = Range.S'$ then at least one x whose value is largest at S has been incremented and its value is still largest at S'. In this case, every x whose value is largest at S' also has the largest value at S. From these facts and from (4) and (5), we conclude

$$(Rank.S = Rank.S') \Rightarrow (\forall \text{ node } i : x.i \text{ is largest at } S \equiv x.i \text{ is largest at } S') \tag{6}$$

Proposition ii follows directly from (3), (4), (5), and (6).

Proof of iii: Consider a computation that starts with a non-unison state S; we show that the value of *Rank* eventually decreases along that computation. From ii, the value of *Rank* does not increase along any computation; therefore, it suffices to show that the value of *Rank* does not remain constant along the computation.

Because the starting state S is a non-unison state, not all x's have equal values at S. Hence, from graph connectivity, the graph has two neighboring nodes i and j such that $x.i$ has the largest value at state S, and $x.i > x.j$ at S. Suppose that the value of *Rank* remains constant along some initial prefix of the computation. Then, from ii, $x.i$ has the largest value and $x.i > x.j$ at every state of the prefix. Along this prefix, procedure *neighbor.i* eventually returns a neighbor v of node i, possibly node j, such that $x.i > x.v$. At the resulting state, say S', neither action of program $P.i$ is enabled. Thus the value of $x.i$ at the next state S'' remains the same as its value at S'. In other words, the largest value of all the x's remains constant in the transition from S' to S''. But because the smallest value of all the x's is guaranteed to increase in every transition (including the one from S' to S''), we have $Range.S' > Range.S''$. From ii, we get $Rank.S' > Rank.S''$. This concludes our proof of iii.

As mentioned earlier, Theorem 1 follows from propositions i and iii. □

Theorem 2: (*Unison*) Starting from any unison state, the value of every x is incremented by 1 in every step.

Proof: At a unison state, all the x's are equal and only the second action of every program $P.i$ is enabled. Executing the assignment statements of all these actions in parallel increments by 1 the value of every x. Thus, the resulting state is a unison state, and the same argument can be repeated to show that the value of every x is incremented in every step. □

4 Convergence Span

The proof of Theorem 1, and especially that of proposition iii, describes how the system converges to a unison state. One of the nodes whose x is largest "eventually" compares its x with the x of a neighboring node, and finds that the value of its x is larger. As a result, this node does not increment its x in the current step, and the value of *Top* is decreased by one. This continues at most $n-1$ times before *Range* is decreased by one.

Let k be the average number of steps it takes a node whose x is largest to finally compare its x with a neighboring x of a lesser value. (A good estimate for k could be the average number of neighbors for each node in the graph.) Starting from a state S, the system converges to unison within

$$k \times (n-1) \times (Range.S) \text{ steps.}$$

Note that if S is a unison state then this value reduces to zero as expected.

5 Related Work

The above system was inspired by the recent work of Even and Rajsbaum [2], in which they consider how to start up a distributed system in the absence of a global start up signal (i.e., one that can be heard by all the programs in the system at the same time). Their work has led to a system in which the program of node i can be defined as follows:

```
begin   (∀ neighbor $v$ of node $i$ : $(x.i = x.v) \vee (x.i = x.v - 1)$)
            → $x.i := x.i + 1$
end
```

This system solves the start up problem as follows. If the system is started at a unison state, but the programs of different nodes start executing at different instants, then although unison may be lost in the early states of the computation, unison is eventually restored as all programs become executing.

There are, however, two problems with this system. First, if the system starts at a non-unison state, then its convergence to a unison state is not guaranteed. For example, if the system is started at a state where the values of two neighboring x's differ by two, then unison can never be reached. Second, the "atomicity" of this system is large: testing the guard of an action at some node requires comparing the x of that node with the x's of all its neighbors.

The first problem can be solved by slightly modifying the program of node i to become:

```
begin (∀ neighbor $v$ of node $i$: $x.i \leq x.v$) $\rightarrow x.i = x.i + 1$ end
```

Our interest in solving the second problem has led to the system in Section 2. It is possible to simplify this system slightly by making the program of node i as follows:

```
begin $x.i \leq x.v \rightarrow x.i, v := x.i + 1, neighbor.i$ end
```

Correctness of the last two systems can be established by an argument similar to the correctness proof in Section 3.

In reviewing this note, one referee observed that if our original program $P.i$ is modified slightly to:

```
begin     $x.i < x.v$    $\rightarrow$  $x.i, v$  $:= x.v + 1,$  $neighbor.i$
      []  $x.i \geq x.v$  $\rightarrow$  $x.i, v$  $:= x.i + 1,$  $neighbor.i$
end
```

then stabilization can be proved easily. In particular, the modified program satisfies the following proposition (which is not satisfied by the original program).

> If a state S is followed by a state S', then
> (∀ node i: $x.i$ is largest at $S \Rightarrow x.i$ is largest at S').

Using this proposition, one can prove a second proposition:

> (∀ neighboring nodes i and j:
> if $x.i$ is largest at some state, then $x.j$ is largest at some subsequent state).

From these two propositions, it is straightforward to show that starting from any state, the system eventually reaches a state where every $x.i$ is largest, such a state is clearly a unison state.

In this program, every $x.i$ is incremented by at least 1 in every step, even when the system is in non-unison. By comparison, our original program does not increment the largest $x.i$'s when the system is in non-unison. Thus, the modified program appears to "consume" the $x.i$'s faster than our original program during non-unison. We are thankful to the referee for his/her keen observation and good taste.

Acknowledgement: We are thankful to Shimon Even for bringing [2] to our attention, and to the Austin Tuesday Afternoon Club, Anish Anora, Nissim Francez, and Jayadev Misra for reading an earlier draft of this note and suggesting some improvements in the presentation.

References:

[1] G. Brown, M. Gouda, and C. Wu, "Token Systems that Self-Stabilize," *IEEE Trans. on Computers*, Vol. 38, No. 6, pp. 845-853, June 1989.

[2] S. Even and S. Rajsbaum, "Lack of a Global Clock does not Slow Down the Computation in Distributed Networks," unpublished manuscript, October 4, 1988.