Time Optimal Self-Stabilizing Synchronization

EXTENDED ABSTRACT

Baruch Awerbuch* Shay Kutten† Yishay Mansour‡ Boaz Patt-Shamir§
George Varghese¶

Abstract

In the network synchronization model, each node maintains a local pulse counter such that the advance of the pulse numbers simulates the advance of a clock in a synchronous network. In this paper we present a time optimal self-stabilizing scheme for network synchronization. Our construction has two parts. First, we give a simple rule by which each node can compute its pulse number as a function of its neighbors' pulse numbers. This rule stabilizes in time bounded by the diameter of the network, it does not invoke global operations, and does not require any additional memory space. However, this rule works correctly only if the pulse numbers may grow unboundedly. The second part of the construction (which is of independent interest in its own right) takes care of this problem. Specifically, we present the first self-stabilizing reset procedure that stabilizes in time proportional to the diameter of the network. This procedure can be combined with unbounded-register protocols to yield bounded-register algorithms.

*Lab. for Computer Science, MIT. Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, DARPA contract N00014-89-J-1988, and a special grant from IBM.

†IBM T.J. Watson Research Center.

‡Tel-Aviv University and IBM T.J. Watson Research Center.

§Lab. for Computer Science, MIT. Research partly done while visiting IBM T.J. Watson Research Center. Supported in part by DARPA contracts N00014-92-J-4033 and N00014-92-J-1799, ONR contract N00014-91-J-1046, and NSF contract 8915206-CCR.

¶DEC, 550 King Street, Littleton, MA 01460.

25th ACM STOC '93-5/93/CA,USA
© 1993 ACM 0-89791-591-7/93/0005/0652...$1.50

1 Introduction

The quintessential problem of virtually every distributed computation is how to remove uncertainty that might effect the task at hand. The nature of distributed systems is such that the tasks may be influenced by many factors; to mention just a few, we need to confront difficulties such as physically dispersed inputs, asynchrony of computation and communication, dynamically changing networks, and many more. It is highly desirable to have some kind of an automatic transformer that allows a protocol designer not to lose generality while assuming a "friendlier" model than the given real-world environment. Despite its desired generality, such a transformer must be cheap, in the sense that its overhead should be kept small so that it is still practical.

In this paper we present a transformer algorithm for reactive protocols that eliminates two of the problematic uncertainties. Specifically, we implement a distributed *pulse counter* at the nodes that simulates (in some precise sense defined below) the advance of a clock in a true synchronous system. Our algorithm has the important feature that it starts operating correctly *regardless of its initial state* (algorithms of this kind are called "self-stabilizing"). In practice, this means that the algorithm adjusts itself automatically to any change in the network or any unpredictable fault of its components, so long as the faults stop for some sufficiently long period. Our algorithm is simple and easy to implement. The algorithm stabilizes in time proportional to the diameter of the network, and hence it is optimal in that respect.

Problem Statement. We are given an asynchronous message passing network, and our objective is to implement a *distributed pulse service* at the nodes. The service must provide the node at all time with a *pulse number*, subject to the following conditions.

Synchronization: Any message sent at local pulse i is received at the other endpoint before local pulse $i+1$.

Progress: There exist parameters Δ and ρ (that may depend on the network topology), such that for any time interval of length $t > \Delta$, the number of consecutive pulses generated at every node is at least $\rho(t-\Delta)$. The parameter Δ is called *network slack* and ρ is the *progress rate.*

A self stabilizing protocol is required to satisfy these conditions only after a certain *stabilization time* has elapsed. More intuition and elaborate description of the desired properties from such a service are given in Section 2 below.

Previous Work. Synchronizers were the target of considerable research, following the work of Awerbuch [Awe85]. For example, see [AS88, PU89, AP90, APPS92]. The concept of self-stabilization was introduced by Dijkstra [Dij74]. A few general "stabilizer" schemes that upgrade non-stabilizing protocols to be self-stabilizing have been proposed since. These are typically based on a *reset* procedure that can impose an acceptable state on the system. Such schemes can be combined with the aforementioned synchronizers to obtain a self-stabilizing synchronizer. We refer below to the known stabilizers and their complexity bounds. In [AG90], a pre-specified bound on the diameter of the network is required, and the protocol stabilizes in time quadratic *in that bound.* We remark that in a dynamic system, such a bound is typically significantly larger than the actual diameter. In [AKY90], a self-stabilizing spanning tree construction is given, that can be used as the basis to a stabilizing reset procedure. The stabilization time of that protocol is $O(n^2)$, where n is the actual number of nodes in the system. In [DIM91], a randomized spanning tree protocol is implicitly given. That protocol stabilizes in expected time $O(d \log n)$, where d is the actual diameter of the network. In [APV91, Var92] a general self-stabilizing reset protocol is presented, whose stabilization time is $O(n)$. In [AV91], a self-stabilizing synchronizer is presented. This protocol stabilizes in time linear in a pre-specified *bound* on the diameter of the network.

It is worth mentioning the important work of Spinelli and Gallager [SG89] on topological update in dynamic networks. This algorithm has many attractive features (in addition to its simplicity). The main property of the algorithm is that it stabilizes in time proportional to the diameter. Although not stated as such, it is easy to verify that the algorithm is self-stabilizing. The drawbacks of the algorithm are its large space requirement (which is inherent for the topology update problem), its message complexity (that might be exponential), and the assumption that the node identifiers are in the range $1 \ldots n$.

Our Results. The contribution of this paper is twofold. First, we present a simple new rule for self-stabilizing synchronization of networks, with network slack d (the diameter of the network), and progress rate 1. Our rule does not invoke global operations, stabilizes in time linear in the actual diameter of the network without any prior knowledge, and does not require any additional memory space (other than for the pulse counter). In the course of development of this rule, we obtain some interesting results regarding common synchronization rules, with applications to clock synchronization schemes. We believe that the analysis of the new rule captures some of the inherent properties of synchronization. However, this rule suffers from a serious disadvantage, namely it requires unbounded pulse numbers to ensure correctness. We fix this flaw in our second major result. Specifically, we give the first time-optimal self-stabilizing reset procedure, i.e., a reset protocol that stabilizes in time proportional to the actual diameter of the network. This result is of independent interest in its own right, being the first time-optimal stabilizer. The heart of the reset procedure is a novel self-stabilizing spanning tree algorithm that in diameter time produces a tree with diameter height. The bounded protocol needs unique identifiers of the nodes, and a pre-specified bound on the diameter of the network; the complexity of the space requirement and messages size depends logarithmically on the bound.

Notations and Model of Computation. We model the processor network as a fixed undirected graph $G = (V, E)$. For $u, v \in V$ we denote by $dist(u, v)$ the length of the shortest path between u and v. We follow the notational convention that $n = |V|$, and that $d = diameter(G) = \max_{u,v \in V} \{dist(u, v)\}$. For each node $v \in V$, we denote $\mathcal{N}(v) = \{u : dist(u, v) = 1\}$. In this abstract we assume the model of *unit capacity data links*, in which there is at most one outstanding message in transit on every channel at any given time. (This model was defined and justified [APV91, Var92] as a realistic model for any message passing system with *some* bound on the capacity of the channels.) The message delivery time can be arbitrary, but for the purpose of time analysis we assume that each message is delivered in one time unit. We use the method of *local detection* [AKY90, APV91, Var92], in which each node constantly sends its state to all its neighbors. This enables us to present our protocol in a compact formulation of local *rules.* These rules are functions that take the state of the neighborhood (made available by the underlying local detection mechanism), and output a new state for the node.

Paper Organization. This paper is organized as follows. In Section 2 we develop the requirements from a desirable synchronization scheme, by exploring the disadvantages of some of popular schemes. In Section 3 we specify the new synchronization rule and analyze its complexity. In Section 4 we develop an optimal self-stabilizing reset procedure and explain how to apply it to the synchronization rule to obtain a protocol that works with bounded registers.

2 Requirements and Examples

In this section we consider a few preliminary ideas for synchronization rules. By studying their properties, we develop a set of requirements from a desirable scheme.

Synchronization. The synchronization requirement can be defined as follows (cf. problem statement in Section 1).

Definition 1 *Let $G = (V, E)$ be a graph, and let $P : V \to \mathbf{N}$ be a pulse assignment. We say that the configuration (G, P) is* legal *for node v, denoted legal(v), if for all $u \in \mathcal{N}(v)$ we have $|P(u) - P(v)| \leq 1$. We say that the configuration (G, P) is* legal *if for all $v \in V$, legal(v) holds.*

The idea behind Definition 1 is as follows. In the synchronous setting, all messages sent at pulse i are received by pulse $i + 1$. When we simulate executions of synchronous protocols on an asynchronous network, we do not have a global pulse-producing clock. Rather, we want to maintain the *validity* of the messages sent. This can be done by ensuring the a node sends pulse $i + 1$ messages only after it has received all the pulse i messages from its neighbors. Since message delivery is not simultaneous, there can be a skew of the pulse counters at neighbors, but this skew is allowed to be at most one: if the pulse numbers at two adjacent nodes differ by more than 1, then necessarily the node with the higher pulse number has advanced without receiving all messages of prior pulses. This notion of legal configuration gives rise to the following simple synchronization rule, which is implicit in the α synchronizer of [Awe85].

Rule 1 *(Min Plus One)*

$$P(v) \leftarrow \min_{u \in \mathcal{N}(v)} \{P(u) + 1\}$$

The idea is that whenever the pulse number is changed, the node sends out all the messages of previous rounds which haven't been sent yet.

Stabilization. As is well known, Rule 1 is *stable*, i.e., if the configuration is legal (as in Definition 1), then applying the rule arbitrarily can yield only legal configuration. Notice however, that if the state is not legal, then applying Rule 1 may cause pulse numbers to drop. This is something to worry about, since the regular course of the algorithm requires pulse numbers only to grow. Thus it is conceivable that actions taken in legal neighborhoods interfere with the actions taken in illegal neighborhoods. This intuition is captured by the following theorem.

Theorem 1 *Rule 1 is not self-stabilizing.*

Proof: By a counter example. Consider the pulse configuration of a 10-processor ring depicted in Figure 1 (a). Clearly, the vertical edges indicate illegal state. Consider now the execution described in Figure 1 (a-i), obtained by repeated application of Rule 1. It is readily seen that the last configuration (i) is basically identical to configuration (a), with all the pulse numbers incremented by one, and rotated one step counter-clockwise. Repeating this schedule results in an infinite execution in which each processor takes infinitely many steps, but none of the configurations is legal. ∎

Let us make a make a short digression here. Theorem 1 has an interesting corollary for *clock synchronization*: one of the popular schemes for clock synchronization [LL84] is "repeated averaging". Roughly speaking, in the repeated averaging rule each node sets its value to be the average value of its neighbors, while advancing the clock if this average is close enough to its own value.

Corollary 1 *Repeated averaging does not stabilize.*

Proof Sketch: The scenario in the proof of Theorem 1 shows that averaging with rounding down does not work. A similar scenario can be constructed for averaging with rounding up. ∎

Time Complexity. One idea that can pop into mind to try to repair the above flaw is to never let pulse numbers go down. Formally, the rule is the following.

Rule 2 *(Monotone Min Plus One)*

$$P(v) \leftarrow \max\left\{P(v), \min_{u \in \mathcal{N}(v)} \{P(u) + 1\}\right\}$$

Rule 2 can be proven to be self-stabilizing. However, it suffers from a serious drawback regarding its stabilization time. Consider the configuration depicted in Figure 2.

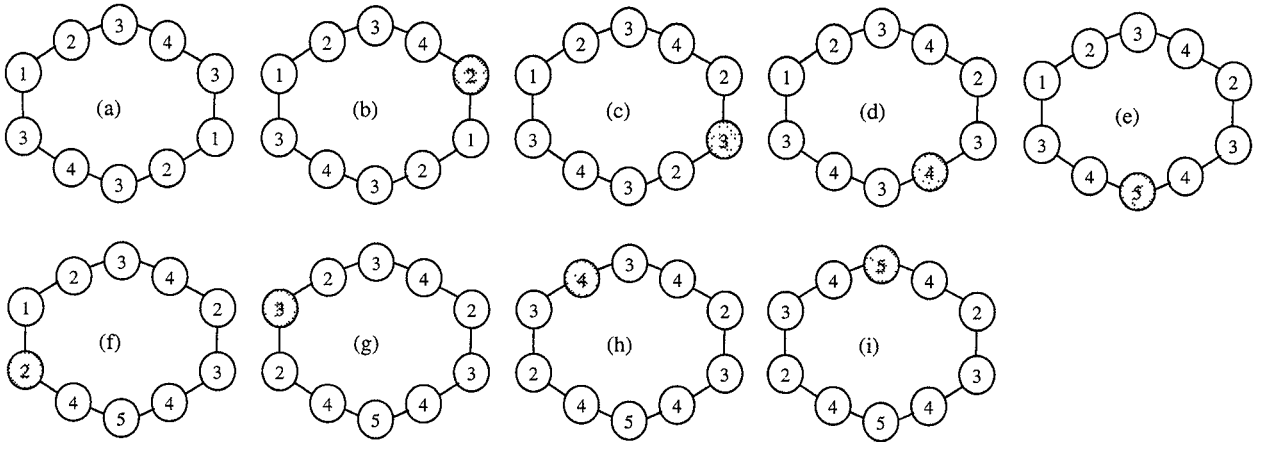

Figure 1: *An execution using Rule 1. The node that moved in each step in marked.*

A quick thought should suffice to convince the reader that the stabilization time for this configuration using Rule 2 is in the order of 1000000 time units, which seems to be unsatisfactory for such a small network. This example demonstrates an important property that we shall require from any self-stabilizing protocol: *the stabilization time must not depend on the initial state; rather, it should be bounded by a function of the network topology.* A clear lower bound on the stabilization time is the diameter of the network (e.g., if $n-1$ nodes must change their pulse number). In the example above, and using Rule 2, the stabilization time depends linearly on the value of the pulses, thus implying that the stabilization time can be arbitrarily large.

Truly Distributed Model. The next idea is to have a combination of rules: certainly, if the neighborhood is legal, then the problem specification requires that Rule 1 is applied. But if the neighborhood is not legal, another rule can be used. The first idea we consider is the following.

Rule 3 *(Maximum)*

$$P(v) \leftarrow \begin{cases} \min_{u \in \mathcal{N}(v)} \{P(u)+1\} , & \textit{if legal}(v) \\ \max_{u \in \mathcal{N}(v)} \{P(u), P(v)\} , & \textit{otherwise} \end{cases}$$

It is straightforward to show that if an atomic action consists of reading one's neighbors and setting its own value (in particular, no neighbor changes its value in the meanwhile), then Rule 3 above indeed converges to a legal configuration. Unfortunately, this model, traditionally called *central demon* model [Dij74, BP89], is not adequate for a truly distributed system, which is based on loosely coordinated asynchronous processes. And as one might suspect, Rule 3 does not work in a truly distributed system, as we demonstrate in the following theorem.

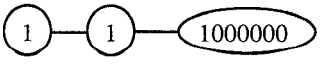

Figure 2: *A pulse assignment for Rule 2.*

Theorem 2 *Rule 3 is not self-stabilizing in an asynchronous system.*

Proof: By a counter example. Consider the execution of a line of 3 processor depicted in Figure 3.

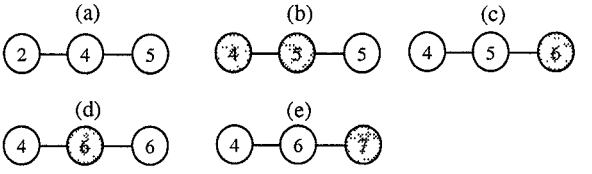

Figure 3: *An execution using Rule 3 in a truly distributed system. The nodes that send or receive are marked.*

Since the estimate of the middle processor of the value of the left processor in configuration (c) is 2, it applies the correcting part of Rule 3, resulting in yet another illegal configuration (d). Since configuration (e) is equivalent to configuration (a) with pulse numbers incremented by 2, we conclude that repeating this schedule results in an execution with infinitely many illegal states. ∎

Locality and Simplicity. Finally, we would like to address two properties which are somewhat harder to capture formally. It seems, however, that these properties are of the highest importance in practice. The

first such property is *locality*. By locality we mean that it is much preferable that a processor will be able operate while introducing only the minimal possible interference with other nodes in the network. In other words, invoking a global operation is considered costly, and we would like to avoid it as much as possible. One way to capture this intuition approximately in our case is to require that the only state information at the nodes is the pulse number, and that protocols should operate by applying local rules as above. The problem with the conventional theoretical approach here is that it neglects to consider the frequent cases, by focusing on the worst case scenario. As an illustrative exercise, contrast this approach with the following solution: whenever an illegal state is detected, *reset* the whole system. This solution, although it may be unavoidable in some rare cases in practice, does not seem particularly appealing as a routinely activated procedure. Consider, for example, the common situation in which a new node joins the system (perhaps it was down for some while). We would like the protocols to feature *graceful joining* in this case, i.e., that other nodes would be effected only if necessary (e.g., the neighbors).

The last property we require from distributed protocols is even harder to define precisely. Essentially, we would like to have the protocols conceptually *simple*. This will make the protocols easy to understand, and therefore, easy to maintain. This requirement is one of the main obstacles for many sophisticated protocols that are not implemented in practice.

3 Optimal Self-Stabilizing Rule

In this section we give a simple, self-stabilizing, optimal rule of synchronization, and analyze its stabilization time. Specifically, our synchronization scheme is based on the following rule.

Rule 4 *(Max Minus One)*

$$P(v) \leftarrow \begin{cases} \min_{u \in \mathcal{N}(v)} \{P(u)+1\} , & \text{if legal}(v) \\ \max_{u \in \mathcal{N}(v)} \{P(u)-1, P(v)\} , & \text{otherwise} \end{cases}$$

In words, Rule 4 says to apply a "minimum plus one" rule (Rule 1) when the neighborhood seems to be in a legal configuration, and if the neighborhood seems to be illegal, to apply a "maximum minus one" rule (but never decrease the pulse number). The similarity to the "maximum" rule (Rule 3) is obvious. The intuition behind the modification is that if nodes change their pulse numbers to be the *maximum* of their neighbors, then "race condition" might evolve, where nodes with high pulses can "run away" from nodes with low pulses. If the correction action takes the pulse number to be one less than the maximum, then the high nodes are "locked", in the sense that they cannot increment their pulse counters until all their neighborhood have reached their pulse number. This "locking" spreads automatically in all the "infected" area of the network. Formally, the way Rule 4 corrects any initial state is analyzed in detail in the proof of Theorem 3 below. We remark that the analysis presented here is simplified (the case analysis of the interleaving due to asynchrony is omitted). This is done to improve exposition of the central ideas.

Theorem 3 *Let $G = (V, E)$ be a graph with diameter d, and let $P : V \to \mathbf{N}$ be a pulse assignment. Then applying Rule 4 above results in a legal configuration in d time units.*

In order to prove Theorem 3, we shall need some tools to analyze the behavior of the synchronization scheme. The basic concept that we use is a certain *potential* value we associate with every node, described in the following definition.

Definition 2 *Let v be a node in the graph. The* potential *of v is denoted by $\phi(v)$ and is defined by*

$$\phi(v) = \max_{u \in V} \{P(u) - P(v) - dist(u, v)\} .$$

Intuitively, $\phi(v)$ is a measure of how much is v *not* synchronized, or alternatively the size of the largest distance-adjusted skew in the synchronization of v. Pictorially, one can think that every node u is a point on a plane where the x-coordinate represents the distance of u from v, and the y coordinate represents the pulse numbers (see Figure 4 for an example). In this representation, v is the only node on the y-axis, and $\phi(v)$ is the maximal vertical distance of any point (i.e., node) above the 45-degree line going through $(0, P(v))$.

Let us start with some properties of ϕ, whose (trivial) proofs are omitted.

Lemma 2 *For all nodes $v \in V$, $\phi(v) \geq 0$.*

Lemma 3 *A configuration of the system is legal if and only if for all $v \in V$, $\phi(v) = 0$.*

We now show the key property of Rule 4, namely that the potential of the nodes never increases when Rule 4 is applied.

Lemma 4 *Let P be any pulse assignment, and suppose that some node u changes its pulse number by applying Rule 4. Denote the new pulse number of u by $P'(u)$, and the potential of the nodes in the new configuration by ϕ'. Then for all nodes $v \in V$, $\phi'(v) \leq \phi(v)$.*

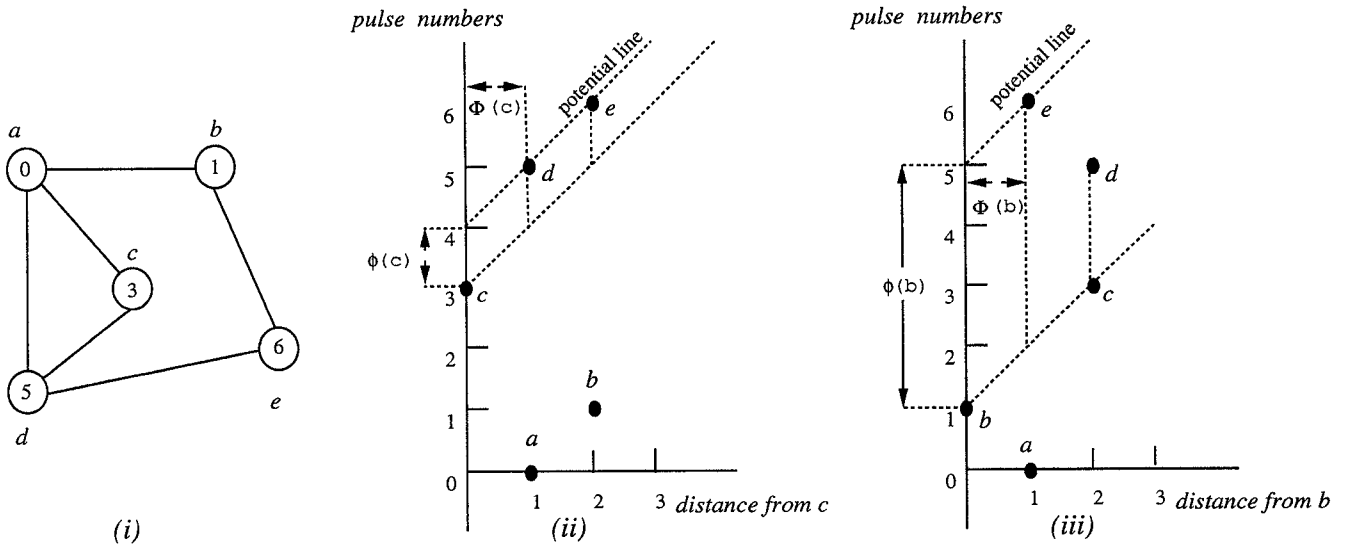

Figure 4: *On the left is an example of a graph with pulse assignment (i). Geometrical representations of this configuration are shown in (ii) and (iii). The plane corresponding to node c is in the middle (ii), and the plane corresponding to node b is on the right (iii). As can be readily seen,* $\phi(c) = 1$, *and* $\phi(b) = 4$. *Also,* $\Phi(c) = 1$, *and* $\Phi(b) = 1$ *(see Definition 3).*

Proof Sketch: The first easy case to consider is the potential of u itself. Since $P'(u) > P(u)$, we have

$$\begin{aligned} \phi'(u) &= \max_{w \in V} \{P(w) - P'(u) - dist(w,u)\} \\ &\leq \max_{w \in V} \{P(w) - P(u) - dist(w,u)\} \quad (1) \\ &= \phi(u) \ . \end{aligned}$$

(Note, for later reference, that the inequality in (1) is strict if $\phi(u) > 0$.) Now consider $v \neq u$. The only value that was changed in the set

$$\{P(w) - P(v) - dist(w,v) \mid w \in V\}$$

is $P(u) - P(v) - dist(u,v)$. There are two cases to examine. If u changed its pulse by applying the "min plus one" part of the rule, then there must be a node w which is a neighbor of u, and is closer to v, i.e., $dist(u,v) = dist(w,v)+1$. Also, since "min plus one" was applied, we have $P'(u) \leq P(w) + 1$. Now,

$$\begin{aligned} P'(u) &- P(v) - dist(u,v) \\ &\leq (P(w)+1) - P(v) - (dist(w,v)+1) \\ &= P(w) - P(v) - dist(w,v) \end{aligned}$$

and hence the $\phi(v)$ does not increase in this case. The second case to consider is when u has changed its value by applying the "max minus one" part of the rule. The reasoning is dual to the first case: let w be a neighbor of u with $P(w) = P'(u)+1$. Clearly, $dist(w,v) \leq dist(u,v) + 1$. This implies that

$$\begin{aligned} P'(u) &- P(v) - dist(u,v) \\ &\leq (P(w)-1) - P(v) - (dist(w,v)-1) \\ &= P(w) - P(v) - dist(w,v) \end{aligned}$$

and we are done. ∎

As noted above, the inequality in (1) is strict if $\phi(u) > 0$. In other words, each time a node with positive potential changes its pulse number, its potential decreases. This fact, when combined with Lemmas 2 and 3, immediately implies eventual stabilization. However, this argument leads to a proof that the stabilization time is bounded by the total potential of the configuration, which in turn depends on the initial pulse assignment. We need a stronger argument in order to prove a bound on the stabilization time that depends only on the topology, as asserted in the statement of Theorem 3. Toward this end, we define the notion of "wavefront".

Definition 3 *Let v be any node. The* wavefront *of v, denoted $\Phi(v)$, is defined by*

$$\Phi(v) = \min_{u \in V} \{dist(u,v) \mid P(u) - P(v) - dist(u,v) = \phi(v)\} \ .$$

In the graphical representation, the wavefront of a node is simply the distance to the closest node of on the "potential line" (see Figure 4 for an example). Intuitively, one can think of $\Phi(v)$ as the distance to

the "closest largest trouble" of v. The importance of the wavefront becomes apparent in Lemma 6 below, but let us first state an immediate property it has.

Lemma 5 *Let $v \in V$. Then $\Phi(v) = 0$ if and only if $\phi(v) = 0$.*

Lemma 6 *Let v be any node with $\Phi(v) > 0$, and let $\Phi'(v)$ be the wavefront of v after one time unit. Then $\Phi'(v) \leq \Phi(v) - 1$.*

Proof: Suppose $\Phi(v) = f > 0$ at some state. Let u be any node such that $P(u) - P(v) - dist(u, v) = \phi(v)$, and $dist(u, v) = f$. Consider a neighbor w of u which is closer to v, i.e., $dist(w, v) = f - 1$ (it may be the case the $w = v$). From the definition of $\Phi(v)$, it follows that $P(w) < P(u) - 1$. Now consider the next time in which w applies Rule 4. If at that time $\Phi(v) < f$, we are done. Otherwise, w must assign $P(w) \leftarrow P(u) - 1$. No greater value can be assigned, or otherwise Lemma 4 would be violated. At this time, $P(w) - P(v) - dist(w, v) = \phi(v)$ also, and hence $\Phi(v) \leq f - 1$. ∎

The next corollary follows from Lemmas 5 and 6.

Corollary 7 *Let v be any node. Then after $\Phi(v)$ time units, $\phi(v) = 0$.*

We can now prove Theorem 3. We first re-state it.

Theorem 3 *Let $G = (V, E)$ be a graph with diameter d, and let $P : V \rightarrow \mathbf{N}$ be a pulse assignment. Applying Rule 4 above results in a legal configuration in d time units.*

Proof: By Lemma 3, it suffices to show that after d time units, $\phi(v) = 0$ for all $v \in V$. From Corollary 7 above, we actually know that a slightly stronger fact holds: for all node $v \in V$, after $\Phi(v)$ time units, $\phi(v) = 0$. The theorem follows from the facts that for all $v \in V$, $\Phi(v) \leq d$, and by the fact that $\phi(v)$ never increases, by Lemma 4. ∎

4 Stabilization with Bounded Registers

In this section we propose a general scheme for stabilizing unbounded-values algorithm that are implemented with realistic (and therefore, bounded-size) registers. Our scheme is based on the following idea, which is described in detail in [APV92]. First, we let the registers be of size large enough so as to accommodate normal operation when initialized at some default value. The crucial part of the scheme is that whenever some processor hits the bound of values which the register is capable of storing, it invokes a *reset* protocol. Roughly speaking, the effect of this procedure is to eventually supply all the nodes in the system with a "signal", such that these signals constitute a consistent reference point in time, in which the nodes can reset their local state and start the computation anew. The justification for the usage of such a costly global operation is that it is invoked *rarely*.

The best implementation of a self-stabilizing reset protocol to date is given in [APV91]. The stabilization time of this procedure can be bounded only by the length of the longest simple path in the network, i.e., $O(n)$ for general networks. In this section we give the first implementation of a reset procedure that works in diameter time.

The idea is to construct a subgraph whose longest simple path has length $O(d)$. Then, when the reset procedure of [APV91] is applied only on the links of that subgraph, we get a reset protocol that stabilizes in $O(d)$ time. Thus, the problem is reduced to the construction of such a subgraph. In the remainder of this section we develop a simple algorithm that produces a *shortest paths* tree rooted at some node in the network. Since the longest simple path in such a tree consists of no more than $2d$ links, in this we will complete our construction.

4.1 Basic Protocols

It is fairly safe to say that one of the first distributed self-stabilizing algorithms (although it was never introduced as such) is the Bellman-Ford algorithm for shortest paths [Bel58]. In the shortest paths problem, there is a designated source node s, and each node v has a variable d_v, called the *distance estimate* of v. The goal of the algorithm is that d_v will hold the actual distance of v from s. We shall consider here only the simple case of unweighted edges. The algorithm can be expressed in our formulation by the following rule.

Rule 5 *(Bellman-Ford)*

$$d_v \leftarrow \begin{cases} 0, & \text{if } v = s \\ 1 + \min\{d_u : u \in \mathcal{N}(v)\}, & \text{otherwise} \end{cases}$$

Rule 5 can be extended easily to produce, at each node, a pointer to the neighbor on the shortest path to s, thus constructing a *shortest paths spanning tree.* (The extension only involves introducing at each node a consistent tie breaker among its incident edges.) It is straightforward to verify, by induction on distance, that if all nodes start with distance estimate which is not smaller than their true distance from v, then

their estimate stabilizes on the true value in time proportional the diameter of the graph. It is slightly less obvious, but nevertheless true, that this holds also for initial estimates which are smaller than the true distance. Informally, the reason for this is that the nodes keep verifying that their distance estimate has some neighbor with an estimate that "supports" it, and therefore if a node $v \neq s$ has $d_v < d_u$ for all $u \in \mathcal{N}(v)$, its distance estimate will increase, until it reaches the true estimate which is supported by a solid "flow" of values from s.

It is important to notice that assuming the existence of a designated source in a distributed system is painful. More formally, this means that we assume that *leader election* can be done; unfortunately, leader election in the context of dynamically changing networks is an equivalently difficult task. The common solution for this problem is to assume that each node has a unique ID (this is a more reasonable assumption to accomplish in practice, and it is actually the industry standard). The idea now is to construct the shortest paths tree rooted at the node with the *minimal ID* in the network. This is done by labeling each distance estimate with its alleged source, and letting smaller IDs always take precedence over larger ID. Technically, the implementation is as follows. Each node v maintains a pair (r_v, d_v), where r_v denotes the minimal ID seen so far (the "alleged root"), and d_v is the distance estimate to r_v. We assume that there is a distinct ID "hardwired" in every node v. The nodes repeatedly apply the following rule.

Rule 6 *(Bellman-Ford with IDs)*
First, a node v computes $r_v \leftarrow \min\{ID_v, r_u\}$, where $u \in \mathcal{N}(v)$. Then v computes d_v by setting $d_v \leftarrow 1 + \min\{d_u : u \in \mathcal{N}(v)$ and $r_v = r_u\}$ if $r_v \neq ID_v$, or $d_v \leftarrow 0$ if $r_v = ID_v$.

Let us remark first that Rule 6 above is the first rule throughout this discussion that violates the locality condition in that it has an extra state component, namely r_v. But Rule 6 has more serious problems. The fact that a small ID overrides all other IDs makes this rule vulnerable to bad initial assignments. Consider, for example, the case in which the system is initialized in a way such that at some node v there is an alleged root $r_v = r^*$, where r^* is not the ID of any node. Assume further that r^* is smaller than all the actual IDs in the system. In this case, applying Rule 6 must eventually results in a state in which $r_v = r^*$ for all $v \in V$. Notice, however, that thereafter the distance estimates at the nodes will grow unboundedly, since unlike the case of Rule 5, there is no true source to halt the increase of the d_v variables. (Another way to see this is by noticing that the true distance from any node to r^* is infinity.) We call such a bad case *ghost root*. Let us remark here that the ghost root phenomenon is fairly frequent: it happens whenever the node with the minimal ID crashes.

A common fix to this widely used rule [MRR80, AG90], is to parameterize the protocol with some hardwired *bound* on the maximal distance estimate. The nodes simply do not consider any estimate that may cause them to break the bound. More precisely, denote the pre-specified bound by D. D is called the *bound parameter* of the protocol. The modified rule is as follows.

Rule 7' *(Bellman-Ford with IDs and Bound Parameter D)*
First, a node v computes $r_v \leftarrow \min\{ID_v, r_u\}$ where $u \in \mathcal{N}(v)$ and $d_u < D$. Then v computes d_v by setting $d_v \leftarrow 1 + \min\{d_u : u \in \mathcal{N}(v)$ and $r_v = r_u\}$ if $r_v \neq ID_v$, or $d_v \leftarrow 0$ if $r_v = ID_v$.

It is not difficult to show that the effect of ghost roots cannot last more then D time units, and therefore Rule 7 stabilizes in $O(D)$ time (see Lemma 8 below). Therefore, if the bound parameter D is close to the actual diameter d, Rule 7 is close to optimal. But unfortunately, having such a estimate of the diameter is an unrealistic assumption in a world of dynamically changing networks, which is our ultimate goal. Typically, D is at least n, the number of nodes in the network, and it might be significantly larger. Notice that we must have some bound on the number of nodes: otherwise, unique IDs would have been impossible to get. Also, coming up with *some* bound is pretty easy: 2^{64} is a good bound on the diameter of all networks in the foreseeable future. However, the maximal dependence on such huge bounds we are willing to tolerate is *polylogarithmic* (e.g., the size of the registers allocated to hold node IDs).

4.2 The New Protocol

We are now ready to present the new protocol for self-stabilizing spanning tree. The stabilization time of the tree is $O(d)$ time units, provided that a bound D on the diameter is known in advance. We remark that the size of the messages in this algorithm is $O(k \log D)$, where k is the size of the IDs.

The idea is as follows. Assume that we are given some bound D on the diameter of the network whose true diameter is d, which may be significantly smaller. As argued above, such a bound is easy to get. To stabilize in time proportional to d, we run $\log D + 1$ independent "versions" of Rule 7 *in parallel*, where in version i, $0 \leq i \leq \log D$, we set the bound parameter to be 2^i. Before we proceed to explain how we use the

results of these independent versions, let us consider the executions of each version separately.

First, consider the versions whose bound parameter is larger than the true diameter d. For these versions, as mentioned above, we have the following property.

Lemma 8 *If $2^i \geq d$, then version i stabilizes in $O(2^i)$ time units.*

Now consider versions number i such that their bound parameter 2^i is smaller than d. Perhaps surprisingly, these versions do not necessarily stabilize in time proportional to their bound parameter, and not even in time proportional to the true diameter. Their stabilization time can be as high as $\Theta(n)$. To see this, consider the graph with initial parent assignment depicted in Figure 5 (a).

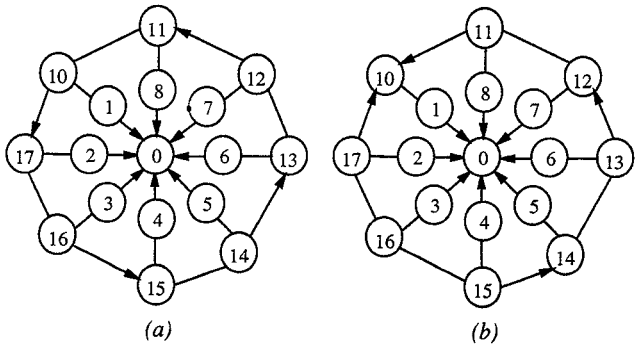

Figure 5: *An initial state for version $i = 0$ (a). The numbers in the nodes indicate their ID, and the arrows indicate their parent pointer. The stable final configuration is depicted in (b).*

Notice that with the given IDs assignment, the stable configuration is the one depicted in Figure 5 (b). But since the center of this graph is "occupied" by a small ID, the information must propagate on the perimeter. It is straightforward to generalize this example to arbitrarily large n while keeping the diameter 4, and thus obtain a lower bound of $\Omega(n)$ time on the stabilization time. (The $O(n)$ upper bound follows from the fact that the sum of the heights of trees in any forest subgraph is $O(n)$.)

The crucial insight needed here is that actually *we do not need lower versions to stabilize.* All we really need is that lower versions will "know" that they do not yield a tree that covers all the network. And this can be done fairly easily, using the following observation: if a version number i satisfies $2^i < d$, then for any tree of version i, at all times, there is at least one node which is in the *fringe* of the tree, i.e., has a neighbor with a different alleged root. These fringe nodes can therefore detect that their tree does not cover the whole network. Hence, the only thing we need now is to let all other nodes of this tree know that version i is actually void. This can be done using the standard technique of broadcast on trees. More specifically, each node v maintains two bits, which we call d_cover_v and u_cover_v. Informally, the u_cover bit is used to propagate information up the tree, by taking repeated logical *and* of the u_cover bits of the children of the node, and the d_cover bit is used to propagate information down the tree, by copying the d_cover bit of the parent. Below, we give the formal rules for the d_cover and u_cover bits. We denote the the parent of v by $parent_v$, and the set of children of v by $child_v$.

Rule 8

$$u\_cover_v \leftarrow \begin{cases} 1, & \text{if } child_v = \emptyset \text{ and } \forall u \in \mathcal{N}(v),\ r_u = r_v \\ \bigwedge_{u \in child_v} u\_cover_u & \text{if } child_v \neq \emptyset \text{ and } \forall u \in \mathcal{N}(v),\ r_u = r_v \\ 0, & \text{otherwise} \end{cases}$$

$$d\_cover_v \leftarrow \begin{cases} d\_cover_{parent_v}, & \text{if } r_v \neq v \\ u\_cover_v, & \text{if } r_v = v \end{cases}$$

For Rule 8 we have the following lemmas.

Lemma 9 *If $2^i < d$, then after $O(2^i)$ time units, at all nodes v, $d\_cover_v = 0$ for version i.*

Lemma 10 *If $2^i \geq d$, then after $O(2^i)$ time units, at all nodes v, $d\_cover_v = 1$ for version i.*

We can now specify the complete protocol. We run $\log D + 1$ versions in parallel. Version i, $0 \leq i \leq \log D$, executes Rule 7 with bound parameter 2^i, and executes also Rule 8. Every version thus maintains its d_cover bit. A node v selects its output by finding the minimal i such that $d\_cover_v = 1$ for version i. The tree edges of that version are the output of the combined protocol.

The combination of Lemmas 8, 9, and 10 gives the following theorem.

Theorem 4 *In $O(d)$ time units, the algorithm produces a shortest paths tree rooted at the minimal ID node in the network.*

As a final remark, let us point out again an interesting property of the algorithm. Although the output of the algorithm stabilizes after $O(d)$ time units, the *state* of the algorithm does not. In particular, for the low versions (with $2^i < d$), full stabilization is guaranteed to occur only after $O(n)$ time; and for the high versions, the stabilization may take as long as $O(D)$ time. But almost magically, the *relevant portion* of the system stabilizes in $O(d)$ time.

References

[AG90] Anish Arora and Mohamed G. Gouda. Distributed reset. In *Proc. 10th Conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 316–331. Spinger-Verlag (LNCS 472), 1990.

[AKY90] Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self-stabilization on general networks. In *Proc. 4th Workshop on Distributed Algorithms*, pages 15–28, Italy, September 1990. Springer-Verlag (LNCS 486).

[AP90] Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *31st Annual Symposium on Foundations of Computer Science*, 1990.

[APPS92] Baruch Awerbuch, Boaz Patt-Shamir, David Peleg, and Mike Saks. Adapting to asynchronous dynamic networks. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 557–570, May 1992.

[APV91] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *32nd Annual Symposium on Foundations of Computer Science*, pages 268–277, October 1991.

[APV92] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilizing network protocols. Unpublished manuscript, 1992.

[AS88] Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks. In *29th Annual Symposium on Foundations of Computer Science*, pages 206–220, October 1988.

[AV91] Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *32nd Annual Symposium on Foundations of Computer Science*, pages 258–267, October 1991.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, October 1985.

[Bel58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[BP89] J.E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, 1989.

[Dij74] Edsger W. Dijkstra. Self stabilization in spite of distributed control. *Comm. ACM*, 17:643–644, 1974.

[DIM91] Shlomo Dolev, Amos Israeli, and Shlomo Moran. Uniform self-stabilizing leader election. In *Proc. 5th Workshop on Distributed Algorithms*, pages 167–180, 1991.

[LL84] J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2-3):190–204, 1984.

[MRR80] John McQuillan, Ira Richer, and Eric Rosen. The new routing algorithm for the ARPANET. *IEEE Trans. Comm.*, 28(5):711–719, May 1980.

[PU89] David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. *SIAM J. Comput.*, 18(2):740–747, 1989.

[SG89] John M. Spinelli and Robert G. Gallager. Broadcasting topology information in computer networks. *IEEE Trans. Comm.*, May 1989.

[Var92] George Varghese. *Self-Stabilization by Local Checking and Correction.* PhD thesis, MIT Lab. for Computer Science, 1992.