Optimal Time Self Stabilization in Dynamic Systems*

(Preliminary Version)

Shlomi Dolev

Dept. of Computer Science, Texas A&M University, College Station, Texas 77843,
e-mail: shlomi@cs.tamu.edu.

Abstract. A *self-stabilizing* system is a distributed system which can tolerate *any number* and *any type* of faults in the history. After the last fault occurs the system starts to converge to a *legitimate behavior*. The self-stabilization property is very useful for systems in which processors may malfunction for a while and then recover. When there is a long enough period during which no processor malfunctions the system stabilizes.

Dynamic systems are systems in which communication links and processors may fail and recover during normal operation. Such failures could cause partitioning of the system communication graph. The application of self-stabilizing protocols to dynamic systems is natural. Following the last topology change each connected component of the system stabilizes independently.

We present *time optimal* self-stabilizing dynamic protocols for a variety of tasks including: routing, leader election and topology update. The protocol for each of those tasks stabilizes in $\Theta(d)$ time, where d is the *actual* diameter of the system.

1 Introduction

A *self-stabilizing* system is a distributed system that can be started in any *possible* global state. Once started the system regains its consistency by itself. Thus, a self-stabilizing system can tolerate *any number* and *any type* of faults in the history. After the last fault occurs the system starts to converge to *legitimate* behavior. The self-stabilization property is very useful for systems in which processors may malfunction for a while and then restart operation in an arbitrary state. When there is a long enough period during which no processor malfunctions the system stabilizes.

Dynamic systems are systems in which communication links and processors may fail and come up during normal operation. One implication of such failure pattern is partitioning of the system into some *connected components*. We are interested in protocols that are resilient to such a partition. Thus, we would not

* This work was supported by NSF Presidential Young Investigator Award CCR-91-58478 and funds from the Texas A&M University College of Engineering.

like our protocol to depend on the existence of a special processor as many self-stabilizing protocols do. Similarly, we require a dynamic protocol not to depend on a-priori knowledge either of the actual diameter or of the actual number of the processors in the connected component. The only a-priori knowledge used is an upper bound, N, on the number of processors in the entire system.

Dynamic networks may be subject to frequent topology changes, hence it is essential for a dynamic protocol to be *time* efficient. The time needed to transport information from one side of a connected component to the other side is no less than $\Omega(d)$, where d is the actual diameter of the connected component. In this paper we present self stabilizing protocols that stabilize in $\Theta(d)$ time.

The main protocol discussed in this paper is the *routing* protocol. The task of the routing protocol is to inform each processor P of the exact set of processors in P's connected component. In addition the protocol gives P the following information about each processor Q in P's connected component: (1) The *distance* (the minimal number of hops) from P to Q. (2) The identifier of a neighbor of P that belongs to a shortest path from P to Q.

Although we stated the above as requirements for the routing task, they could have other interpretations, e.g.: once every processor knows the exact set of processors that belong to its connected component then there is a unique leader for that component – the processor with the maximal[2] identifier.

The study of self-stabilizing systems started with the fundamental paper of Dijkstra [Di-74]. Following the pioneering work of Dijkstra a great amount of work has been done in this area; a partial list of those papers is: [Kr-79], [La-86], [BGW-87], [BP-88], and [DIM-90]. Spinelli and Gallager [SG89] presented a dynamic self-stabilizing protocol for topology update that stabilizes within $\Theta(d)$ time (as a version of their event-driven SPTA protocol). In their protocol each processor P maintains information on the topology, including the distance of every link from P, and communicates this information to all its neighbors. Consequently, the memory that a processor should be equipped with is at least $O(k|E|)$, where k is the size of a processor identifier[3], and $|E|$, the number of links in the system ($|E|$ is bounded from above by N^2). Furthermore, $O(k|E|)$ bits are communicated through every communication link.

The memory size required by each processor for our routing protocol is bounded by $O(kN)$ bits. This improves the amount of memory needed for optimal time self-stabilizing leader election, routing and many other tasks w.r.t. to the solution of [SG89].

The protocols we present are *memory adaptive* as defined in [AEV-92]: at some point after the system stabilizes the size of the memory used by each processor, P, is $O(kn)$ where n is the *actual* number of processors in P's connected component. Independently, in [AK+-93] a self stabilizing spanning tree protocol that stabilizes in $\Theta(d)$ time is presented. For the leader election task the space complexity of their protocol is often better than ours. They use $O(k \log D)$ bits where D is a bound on the diameter. In case $\log D$ (in dynamic network the

[2] Another convention could be the minimal identifier.

[3] In most cases a word of memory is sufficient to represent such an identifier

bound on the diameter D is theoretically identical to N) is greater than n our adaptive protocol eventually uses less memory. Moreover, for the case of the routing task our protocol achieves the optimal space complexity[4].

Our protocol may also be used to improve the amount of communication needed for topology update w.r.t. the protocol of [SG89]. Following $O(d)$ rounds the topology (i.e. $O(k|E|)$ bits) is communicated only through links that belong to a (single) spanning tree (i.e. exactly $n-1$ links) while $O(kN)$ bits are communicated through all other links.

In [AV-91] the following open question was stated: "The transformation adds $O(D)$ overhead to the time complexity of the protocol, where D is the bound on the diameter of the network after arbitrary failures. Clearly D can be much larger than the actual diameter of the final network. A natural open problem is to obtain a compiler whose time overhead only depends on the actual diameter of the final network." We close this problem for the case that processors have distinct identifiers by the use of our routing protocol (or the self-stabilizing topology update protocol of [SG89]) that stabilizes in $\Theta(d)$ time and the use of the *fair protocols composition* technique presented in [DIM-90][5]. Independently, in [AK+-93] this open question is closed too.

The rest of the paper is organized as follows. In Section 2 we describe the requirements for a self-stabilizing dynamic distributed protocol. Section 3 contains the self-stabilizing routing protocol which is a composition of a multiple breadth-first search trees (abbreviated BFS trees) protocol and counting protocol. Concluding remarks are in Section 4.

2 Distributed System

A distributed system consists of N processors denoted by $P_1, P_2, \cdots, P_N$. The processors have distinct identifier in the range 1 to $2^k \geq N$. [6] The identity of a processor P is denoted by id_P. Each processor can communicate with some subset of the processors called its *neighbors*. Communication among neighboring processors is carried out by *communication registers*. The atomic operations that these registers support are *read* and *write*. A processor, P, communicates with all its neighbors by using a single writer, multi-reader register in which only P writes and from which all the neighbors of P read. The registers are serializable with respect to read and write actions. The system's *communication graph* is the graph formed by representing each processor as a node and by drawing an (undirected) edge between every two neighbors. In dynamic systems the system's communication graph is not necessarily connected. Since there is no possible communication between two distinct connected components, we consider

[4] Our protocol use the amount of memory a processor needs in order to store the distances to every other processor.

[5] See [DIM-91b] for how to close this open question by the use of randomization in the case that processors do not have distinct identifiers.

[6] Usually the range of the identifiers is much larger than the bound on the number of processors.

each connected component as an independent system that contains a subset of the processors and edges of the original system (in the sequel we use the term connected component and system interchangeably).

For ease of presentation we regard each processor as a CPU whose program is composed of *atomic steps*. An atomic step of a processor consists of an internal computation followed by read or write. We assume that the state of a processor fully describes its internal state and the value written in its register. Denote by S_i the set of states of P_i. A *configuration*, $c \in (S_1 \times S_2 \times \cdots S_n)$ of the system is a vector of states of all processors.

Processor activity is managed by a scheduler. In any given configuration the scheduler activates a single processor which executes a single atomic step. To ensure correctness of the protocols, we regard the scheduler as an adversary. A run of the system is a finite or infinite sequence of configurations $R = (c_1, c_2, \cdots)$ such that for $i = 1, 2, \cdots$, c_{i+1} is reached from c_i by a single atomic step of some processor. A *fair run* is an infinite run in which every processor executes atomic steps infinitely often.

For ease of description, we assume that the computation of each processor P is done in *cycles*. Each *cycle* consists of a sequence of steps in which P executes the following in the specified order: (a) reads the values of the registers owned by its neighbors and (b) writes into the register it owns. Note that by the nature of self-stabilizing systems a processor could start its operation with any atomic step of the above cycle. For instance a processor P could start operation by reading from its second neighbor; however, after P writes in its own register, P starts a new cycle.

In a distributed system each processor may execute atomic steps at any constant or non-constant rate. Various processors might be slow in various parts of the run. The following definition of *round* complexity captures the rate of action of the slowest processor in any segment of the run. Given a run R the first round of R is finished immediately after each processor has executed one cycle; the second round is finished after each processor has executed one cycle following the termination of the first round, and so on and so forth. For any given run, R, the *round complexity* (which is sometimes called the run time) of R is the number of rounds in R.

We proceed by defining the self-stabilization requirements for distributed systems. A behavior of a system is specified by a set of runs. Define a *task* LR to be a set of runs which are called *legitimate runs*. A configuration c is *safe* with respect to a task LR and a protocol PR if *any* fair execution of PR starting from c belongs to LR. Finally, a protocol PR is *self-stabilizing* for a task LR, if starting with any system configuration and considering any fair scheduler, the protocol reaches a safe configuration.

Although the description of the system is similar to [DIM-90] the protocols we present work also in message passing systems by the use of a self-stabilizing simulation of shared memory described in (the full version of) [DIM-91a].

3 The Routing Protocol

In this section we present a dynamic self-stabilizing routing protocol. First we describe a *multiple BFS trees* protocol that constructs n directed BFS trees; for every processor P, the protocol constructs a directed BFS tree rooted at P. A self stabilizing protocol that constructs a single BFS tree rooted at a special processor was presented in [DIM-90]. In fact it is shown there that a similar algorithm to the Bellman-Ford [Be-58] algorithm is self-stabilizing. Roughly speaking our multiple BFS trees protocol uses n copies of the protocol that appears in [DIM-90]. Following the description of the multiple BFS trees protocol we present a *counting protocol* that is integrated into the multiple BFS trees to form the routing protocol.

Roughly speaking, after the multiple BFS trees protocol constructs all the BFS trees (in $O(d)$ rounds) the processors in the system might not be aware of the fact that the BFS trees are already fixed; activities that are related to previously connected processors (or due to the arbitrary initial configuration) could still take place. We use the counting protocol to eliminate the influence of these activities within extra $O(d)$ rounds. Each processor uses its (eventually) fixed tree to count the number of processors in the system. The counting part results with the right answer $O(d)$ rounds after the BFS tree is already constructed. Then every processor can distinguish the information that is related to processors in the system from the total information the processor has.

3.1 Multiple BFS Trees Protocol

The multiple BFS trees protocol constructs the *first BFS tree* that is rooted at each processor: a graph may have more than a single BFS tree rooted at the same node. We define the *first BFS tree* (similarly to [DIM-90]) of G relative to P to be a BFS tree, rooted at P. In case a node, Q, of distance $j+1$ from P, has more than one neighbor of distance j from P, Q is connected to the neighbor with the maximal identifier, among all its neighbors whose distance from P is j.[7]

The task of the multiple BFS trees protocol is defined by a set LR of legitimate runs during which every configuration of the system encodes for every processor P the first BFS tree, rooted at P, of the connected component of P.

Data Structures: During the run of the protocol each processor, P, maintains an array, $tab_P[1 : N]$ with N entries, an entry for each (possible) processor Q in the system. P communicates with any of its neighbors Q by writing in tab_P and reading tab_Q. The i'th, $1 \leq i \leq N$, entry of tab_P, $tab_P[i]$, is a triple $\langle tab_P[i].id, tab_P[i].dis, tab_P[i].f \rangle$ (or in short $\langle id, dis, f \rangle$), of the following ranges: id and f are in the range 0 to 2^k and dis in the range 0 to N. After the system stabilizes it holds that:

[7] Other convention (like the *last BFS tree* could be used as well as long as the tree is fixed.

(a) $tab_P[i].id$ is either an identifier of a processor, or 0. Whenever $tab_P[i].id \neq 0$ then,
(b) $tab_P[i].dis$ is the distance from P to the processor with the identifier $tab_P[i].id$, and
(c) $tab_P[i].f$ is the identifier of the father of P in the first BFS tree of the processor with the identifier $tab_P[i].id$.

To compute the values in tab_P P uses internal tables called $new\_tab[1:N]$, $r\_tab[1:N]$ and $update\_tab[1:2N]$. Each entry of the above internal tables is a triple $\langle id, dis, f \rangle$.

Transition Function: Each processor P builds the first BFS tree identified with id_P. This is accomplished by repeatedly writing in $tab_P[1]$ the tuple $\langle id_P, 0, 0 \rangle$. In order to determine the other entries of tab_P, P repeatedly reads the tables tab of all its neighbors; in case P has δ neighbors the number of tuples read by P is δN.

From the δN tuples P reads, P chooses the maximal number of tuples, up to $N-1$ tuples, that fulfill the following restrictions:
(a) There is no tuple with id_P.
(b) For any identifier Q, if $\mathcal{Q}$ is the set of tuples read with the identifier Q, then only a tuple with the smallest dis among $\mathcal{Q}$ could be chosen.
(c) Let $\mathcal{R}$ be the set of tuples left after deleting all tuples due to (a) and (b) above. If the number of tuples in $\mathcal{R}$ is smaller than $N-1$ then P chooses all of them. Otherwise, P sorts the tuples in $\mathcal{R}$ by their dis values, where a tuple with the smallest dis is the first. Then P chooses the first $N-1$ tuples.

Once P chooses the appropriate tuples, P assigns $tab_P[2:N]$ with the chosen tuples. In case P chose less than $N-1$ tuples, P fills the rest of the entries of tab_P with the tuple $\langle 0, N, 0 \rangle$, which is called the *empty tuple*.

The above description uses an array of size $O(kN)$ to store the table read from each neighbor before choosing the tuples. Thus, for a processor with δ neighbors $O(\delta kN)$ space is required. To eliminate the δ factor we choose candidate tuples after each read operation instead of choosing tuples after reading all the tables of the neighbors.

The description of the protocol for a processor P, that has δ neighbors, appears in Fig. 1. During the execution of the protocol each processor, P, repeatedly reads the tables of its neighbors in a cyclic order. Whenever P begins a computation cycle, P initializes the values of the local tables $new\_tab[1:N]$ by the following assignments: $new\_tab[1] := \langle id_P, 0, 0 \rangle$ and $new\_tab[2:N] := \langle 0, N, 0 \rangle$. (This notation of assignment is a short-hand for the assignment $new\_tab[j] := \langle 0, N, 0 \rangle$ for each entry in the range $2 \leq j \leq N$. In the sequel we use this notation for multiple assignment in an array). Following any read operation P assigns the table it just read to $r\_tab[1:N]$ and updates $new\_tab[1:N]$.

Correctness and Complexity Proof: In this subsection we show that within $3d+1$ rounds a safe configuration is reached after which any configuration encodes n first BFS trees, as required.

1. do forever
2. Initialize:
   (a) $new\_tab[1] := \langle id_P, 0, 0\rangle$
   (b) $new\_tab[2:N] := \langle 0, N, 0\rangle$
3. for $i = 1$ to δ do
4. Read tab_Q where Q is P's i'th neighbor:
   (a) $r\_tab[1:N] := read(tab_Q[1:N])$
5. Choose tuples:
   (a) $update\_tab[1:N] := new\_tab[1:N]$
   $update\_tab[N+1:2N].\langle id, dis, f\rangle := \langle r\_tab[1:N].id, r\_tab[1:N].dis + 1, id_Q\rangle$.
   (b) P *deletes* every entry j of $update\_tab$ such that the value of its dis is equal to N. (P deletes an entry in a table by the assignment of the triple $\langle 0, N, 0\rangle$ to it.)
   (c) For every id that appears in more than one entry of $update\_tab$, say in the set of entries I, P finds a subset $I' \subseteq I$ s.t. the values of their dis field are minimal in I. P deletes all the entries that belong to I except the entry with the largest f in I'.
   (d) P sorts (the modified) $update\_tab$ by the value of the dis field of its entries s.t. a tuple with the smallest dis is in $update\_tab[1]$.
   (e) P assigns $new\_tab[1:N] := update\_tab[1:N]$.
   (f) If $i = \delta$ then P writes $tab_P[1:N] := new\_tab[1:N]$.
6. enddo(3)
7. enddo(1)

Fig. 1. Multiple BFS trees Protocol - P's Program.

Lemma 1. *Following the first round of any run it holds for any processor P that:*
(1) $tab_P[1] = \langle id_P, 0, 0\rangle$.
(2) For any $2 \leq j \leq N$ $tab_P[j].dis > 0$.

Proof. During the first round a processor P assigns $new\_tab[1] := \langle id_P, 0, 0\rangle$ (operation 2(a)) assigns $new\_tab[2:N] := \langle 0, N, 0\rangle$ and then $update\_tab[1:N] := new\_tab[1:N]$ (operation 5(a)) at least once. Let c be the configuration that immediately follows those operations. Following each time P executes these assignments P reads tab_Q (operation 4(a)) of every neighbor Q and then writes in tab_P (operation 5(f)). We now show that $new\_tab[1] = update\_tab[1] = \langle id_P, 0, 0\rangle$ following c. new_tab is assigned only during operations 2(a)-2(b) and 5(e). Thus, the only possibility of $new\_tab[1]$ to be changed is due to operation 5(e). Therefore, we only have to prove that $update\_tab[1]$ is never changed. Assume towards contradiction that $update\_tab[1]$ is changed following c. The first change is not due to operation 5(b) since $update\_tab[1].dis = 0$. Now we show that the first change is also not due to 5(c)-5(d). First notice that by 2(a) and 5(a) following c the dis field of every entry of $update\_tab$ but $update\_tab[1]$ has a value that is greater than 0. Thus, $update\_tab[1]$ is the only tuple in the calculated I' for id_P during 5(c) and it is the first tuple due to the sort operation of 5(d). □

Define a *dangling id* in some configuration c to be an *id* that appears in at least one of the tables $tab[1:N]$, $new\_tab[1:N]$ or $r\_tab[1:N]$ but there is no processor with the same *id* in the same connected component of the system. Notice that this definition does not consider tuples in *update_tab*. Those tuples are assigned only to *new_tab* and only during operation 5(e). By the definition of an atomic step if operation 5(e) is executed then operation 5(a) is executed before and during the same atomic step. Hence, by operation 5(a) the source of the tuples in *update_tab* is *new_tab* and *r_tab* of c.

To prove that the system stabilizes in $O(d)$ rounds it is important to prove first that the minimal *dis* value of a dangling *id* grows by at least 1 during every round.

Lemma 2. *In every run, following the first $d+1$ rounds every dis field of an entry of a dangling id in r_tab is at least d and every dis field of an entry of a dangling id in tab or new_tab is greater than d.*

Proof. Let $id_D(c)$ be a dangling *id* in, c, the first configuration of the run. Let $dis_D(c)$ be the minimal *dis* among the tuples in *tab*, *new_tab* or *r_tab* with id_D in c. Every processor executes at least one cycle during the first round. Hence, every processor executes the operations 2(a), 2(b) and 5(a) during the first round. Following the execution of those operation and till the first round is ended every processor writes in *r_tab*, *new_tab* and *tab*. A triple with a dangling *id* could be written in *r_tab*, *new_tab* and *tab* only due to the following operations in the specific order:
(1) read of dangling *id* during operation 4(a).
(2) assignment of a tuple of *r_tab* into *update_tab* (operation 5(a)).
(3) assignment into *new_tab* (operation 5(e)).
By operation 5(a) whenever a tuple is assigned to *update_tab* the *dis* field is set to be greater by 1 than the *dis* read. Thus, for every dangling $id_D(c)$ when the first round is ended there is no tuple with $id_D(c)$ in *r_tab* s.t. its *dis* value is less than $dis_D(c)$ and there is no tuple in *new_tab* or tab_P with $id_D(c)$ s.t. its *dis* is less than or equal to $dis_D(c)$. Moreover, after the first read operation that is executed by every processor following the first round, it holds also that there is no tuple with $id_D(c)$ in *r_tab* s.t. its *dis* value is less than or equal to $dis_D(c)$.

The same arguments holds for a run that starts with the configuration c' that is reached immediately after the first round. Since the minimal value of a *dis* field is 0 and since dis_D is incremented by at least one in every round than following $d+1$ rounds the value of dis_D in at least d in any entry with $id_D(c)$ of *r_tab* and is at least $d+1$ in any entry of *new_tab* and *tab* with $id_D(c)$. □

Lemma 3. *Let R' be the suffix of a run R that immediately follows the first $(d+1)$ rounds of R. For any i, in every configuration of R' that follows the first $2i$ rounds of R' it holds for any processor P, and for every processor Q that:*
(a) if Q is at distance greater than i from P, then any tuple in tab_Q with id_P has a dis that is greater than i.
(b) if Q is at distance less than or equal to i from P, then there exists a single

tuple in tab_Q with id_P. In case $\langle id_P, dis, f\rangle$ exists in tab_Q then:
(1) dis is the distance of Q from P and
(2) f is an identifier of the father of Q in the first BFS tree rooted at P.

Proof. The proof is by induction on $i \leq d$, the distance from P.
Base Case $i = 1$: By lemma 1 following the first round of R' the only table in which the tuple $\langle id_P, 0, 0\rangle$ appears is tab_P (every other tuple with id_P has dis greater than 0). During the second round of R' any processor Q at distance greater than 1 reads the tables of all its neighbors and discovers that there is no tuple with id_P and $dis = 0$. Consequently, Q does not assign a tuple with id_P and $dis < 2$ in tab_Q. This proves assertion (a). To prove assertion (b) we first show that during the second round of R', when any processor Q, at distance 1 from P writes $tab_Q[1:N] := new\_tab[1:N]$, there is an entry with the values $\langle id_P, 1, id_P\rangle$ in $new\_tab[1:N]$. Then we show that at the end of any successive computation cycle of Q, Q assigns $\langle id_P, 1, id_P\rangle$ in tab_P.

During the second round of R', Q reads the table of each of its neighbors. Following the read operation (4(a)) from P, Q assigns $\langle id_P, 1, id_P\rangle$ in $update\_tab$ (operation 5(a)). Since $dis = 1$, the tuple $\langle id_P, 1, id_P\rangle$ is not removed during 5(b). $\langle id_P, 1, id_P\rangle$ is not removed during operation 5(c) either, since by the above argument following the first round of R', Q may read the tuple $\langle id_P, 0, 0\rangle$ only from tab_P. Thus, we have to consider operation 5(d). By lemma 2 during R' all the dangling tuples have dis greater than d. Following operation 5(c) each non-deleted entry in $update\_tab$ has a unique id. Since there are at most N non-dangling tuples in $update\_tab$ it is guaranteed that the tuple $\langle id_P, 1, id_P\rangle$ appears in one of the first N indices of $update\_tab$ and is assigned to $new\_tab[1:N]$ during operation 5(e).

Now consider any further execution of the operations 4(a) and 5(a)-5(e) before the execution of 5(f). First the tuple $\langle id_P, 1, id_P\rangle$ is assigned to $update\_tab$ during the assignment $update\_tab[1:N] := new\_tab[1:N]$ of operation 5(a). As before, this tuple is not deleted from $update\_tab$ and is assigned to new_tab during operation 5(e). Thus, at the end of the first cycle of Q during the second round of R', Q assigns the tuple $\langle id_P, 1, id_P\rangle$ to tab_P (operation 5(f)).

The above arguments hold for any cycle execution of Q that follows the first computation cycle of Q during the second round of R'. Hence, whenever Q executes operation 5(f) Q assigns the tuple $\langle id_P, 1, id_P\rangle$ to tab_P. This completes the proof of assertion (b).
Induction Step: We assume that following the first $2i$ rounds of R' assertions (a) and (b) hold for i. We prove that after two additional rounds assertions (a) and (b) hold for $i + 1$.

By the induction assumption, following the first $2i$ rounds of R' any processor Q that is at distance greater than $i+1$ from P cannot read a tuple s.t. its identifier is id_P and its dis is less than $i + 1$. Thus, during the first cycle that follows the first $2i$ rounds of R' it holds for Q that a tuple with id_P could appear only with dis that is greater than $i + 1$. This proves assertion (a).

Let U be an arbitrary processor at distance $i + 1$ from P and let Q be the neighbor of U with the maximal identifier s.t. Q is at distance i from P.

During the $2i+2$'nd round U initializes *update_tab* (operations 2(a)-2(c)). Then U reads tab_Q (operation 4(a)) and assigns the tuple $\langle id_P, i+1, id_Q \rangle$ to *update_tab* (operation 5(a)). Since $dis = i+1 \leq d < N$, the tuple $\langle id_P, i+1, id_Q \rangle$ is not removed during 5(b). $\langle id_P, i+1, id_Q \rangle$ is not removed during operation 5(c) either since following the first $2i+1$ rounds of R', Q may read a tuple with id_P and $dis = i$ only from tables of processors that are at distance i from P and Q has the maximal identifier among them. Thus, we have to consider operation 5(d). By lemma 2 during R' all the dangling tuples have *dis* that is greater than d. Following operation 5(c) each non-deleted entry in *update_tab* has a unique *id*. Since there are at most N non-dangling tuples in *update_tab* it is guaranteed that the tuple $\langle id_P, i+1, id_Q \rangle$ appears in one of the first N indices of *update_tab* and is assigned to $new\_tab[1:N]$ during operation 5(e).

Now consider any further execution of the operations 4(a) and 5(a)-5(e) by U before the execution of 5(f). First the tuple $\langle id_P, i+1, id_Q \rangle$ is assigned to *update_tab* during the assignment $update\_tab[1:N] := new\_tab[1:N]$ of operation 5(a). As before, this tuple is not deleted from *update_tab* and is assigned to *new_tab* during operation 5(e). The above arguments hold for any cycle execution of U that follows the first $2i+1$ rounds of R'. Hence, whenever U executes operation 5(f) U assigns the tuple $\langle id_P, i+1, id_Q \rangle$ in tab_U. □

Corollary 4. *In any run any configuration that follows the first* $3d+1$ *rounds encodes a first BFS tree for any processor* P.

Proof. The use of lemma 3 with $i = d$ yields that for any processor P, following $3d+1$ rounds, a first BFS tree rooted in P is encoded in the tables *tab* of the processors. □

Note that although following the first $3d+1$ rounds each processor is a root of a first BFS tree there still might be dangling tuples. The value of the *dis* field of those tuples is greater than $3d$ and less than N. Actually the dangling tuples are removed by the regular execution of the protocol only after N rounds (this is derived by arguments similar to the arguments of lemma 2). Thus, following $3d+1$ rounds a processor cannot know the exact number of processors in its connected component and obviously can neither know the topology of the component nor choose a leader.

In the following section we augment the protocol with a *counting* protocol that informs each processor in the connected component of the number of processors in its component. Once a processor P has the right knowledge on the number of processors in P's connected component, say l, P consider the closest (due to the distance fields in tab_P) l processors to be in its connected component. The counting protocol is executed during the regular execution of the protocol and converges in $O(d)$ rounds after the first BFS trees of every processor is constructed.

3.2 Counting Protocol

The counting protocol is an example of a self stabilizing *convergecast* scheme, i.e., a scheme for collecting information in a self stabilizing fashion. It uses the

constructed directed spanning trees of the multiple first BFS trees as an input. Given a first BFS tree, information is repeatedly collected starting from the leaves towards the root. Since the information is collected in a fixed direction, eventually the correct information reaches the root. Consequently, when it is necessary, the collected information could be *broadcast* from the root to every processor in the system by the use of an independent mechanism that repeatedly sends information from any father to its sons. Using the self stabilizing convergecast technique, a self stabilizing topology update protocol that stabilizes within $\Theta(d)$ rounds could be achieved. Since topology update is an over-kill when only the number of processors is needed and since it requires larger amount of memory we hereby present a protocol that collects only the information on the number of processors in the spanning tree.

The task of the counting protocol is defined by a set LR of legitimate runs during which the state of every processor P encodes the value n – the actual number of processors in P's connected component.

To integrate the counting protocol into the multiple BFS protocol we add to any tuple a new field called *count* that contains an integer in the range 0 to N. When the system stabilizes, the value of the *count* field in an entry of tab_P, $\langle id_Q, dis, f, count\rangle$, is the number of processors that are descendants of P in the first BFS tree of Q.

For any processor P, following the construction of the first BFS tree of P whenever a processor Q reads a table of a neighbor U, Q can deduce by the value of the father field of the entry $\langle id_P, dis, f, count\rangle$ of tab_U whether U is a son of Q in the first BFS tree of P. Using the above Q sums up the values of the count fields of its sons, adds to this result the number of its sons, and assigns the final result to the *count* field of the entry $\langle id_P, dis, f, count\rangle$ of tab_Q.
To implement the above we modify the multiple BFS trees protocol as follows:

2(c) $sum[0:N] := 0$

4(b) for $j = 1$ to N do

if $r\_tab[j].f = id_P$ then

$sum[r\_tab[j].id] := min(sum[r\_tab[j].id] + r\_tab[j].count + 1, N)$.

5(f) if $i = \delta$ then P writes $(tab_P[1:N].\langle id, dis, f, count\rangle :=$

$\langle new\_tab[1:N].id, new\_tab[1:N].dis, new\_tab[1:N].f, sum[new\_tab[1:N].id]\rangle)$.

Figure 2: Incorporation of the Counting Protocol- P's Program.

Correctness and Complexity Proof:

Lemma 5. *In any run of the counting protocol, d rounds after all the first BFS trees have been constructed the value of the count field of every tuple with $dis = 0$ is $n - 1$.*

Proof. Let T_P be the first BFS tree rooted at P, as encoded by the system configuration. Define the *height* of a processor Q in T_P to be the maximal number of processors in a path of T_P that starts at a leaf and ends at Q. The correctness and complexity proof of the counting protocol is by induction on the height of the processors. The induction assumption is that if Q is in height less than or equal to h then following h rounds the *count* field of the entry $\langle id_P, dis, f, count\rangle$ of tab_Q holds the number of processors that are descendants of Q in T_P. The induction base is by the fact that every processor that is at height 0 is a leaf. The induction step is derived from the induction assumption for processors at height h and the way processors at height $h+1$ calculate the number of processors that are their descendants. Since the height of a processor in any first BFS tree is bounded by d it holds that following $O(d)$ rounds the *count* field of the root is equal to $n-1$. The above arguments are true for any tree in the system, hence for any processor P, the tuple $\langle id_P, 0, 0, n-1\rangle$ appears in tab_P. □

By the above lemma following $O(d)$ rounds the knowledge of any processor, P, is stabilized to include the following correct values:
(1) The set of processors that appears in tab_P includes every processor in P's connected component.
(2) The value $n-1$ appears in the single entry with $id = id_P$ in tab_P – $\langle id_P, 0, 0, n-1\rangle$.
(3) The first n entries of tab_P include exactly the n identifiers of the processors in P's connected component (with their correct distance from P and the first link in a shortest path to each of them).

By the nature of self-stabilizing protocols a processor can never know whether the system is already stabilized. Roughly speaking, a processor could be awakened (in a state that is) "knowing" wrong information which will eventually stabilize to the correct information during the operation of the protocol. For example, in our protocol a processor P could be started with a wrong value of *count* in the tuple $\langle id_P, 0, 0, count\rangle$ and with arbitrary contents of tab_P. However, although P does not know when the system is stabilized, it holds that following $O(d)$ rounds the knowledge P has about its connected component reflects the reality.

4 Concluding Remarks

We presented a dynamic self-stabilizing protocols that stabilize in $\Theta(d)$ rounds. The protocols presented here are easily modified to yield memory adaptive protocols by using connected-lists with bounded length (no more than N elements) instead of arrays. The modification is straightforward: (1) Omit every entry with the empty tuple, i.e. the tuple $\langle 0, N, 0\rangle$ in *update_tab* and *new_tab* and the tuples $\langle 0, N, 0, count\rangle$ in the tables *tab* and *r_tab*. (2) Omit every entry in the *count* array with value 0. As mentioned before the memory requirement is bounded by $O(kN)$ bits. Using the above, it is easy to see that following $O(N)$ rounds the memory required is only $O(kn)$ bits.

We also described a *convergecast broadcast* technique which could be composed with our routing protocol to yield other tasks such as topology update within $\Theta(d)$ rounds. In the topology update protocol, the information that is convergecast is the local topology (i.e. the identity of the neighbors) of each descendant. To achieve better communication complexity than implied by the protocol of Spinelli and Gallager [SG89] we repeatedly collect information on the topology only through the links of the tree of the elected leader. We combine our routing protocol with a convergecast broadcast mechanism using a *fair protocol composition*, a technique presented in [DIM-90].

The convergecast broadcast mechanism assumes that every processor knows its father and sons in T, the BFS tree of the leader. Note that this assumption is valid after $O(d)$ rounds. The convergecast uses for every processor P a variable up_P in which P writes to its father in T. In case P is a leaf in T, P writes its own local topology in up_P. Otherwise P concatenates the values of the up variables of all its sons in T and its own local topology and writes the result in up_P. At the same time in order to inform every processor with the collected topology we repeatedly broadcast the value of the up variable of the leader through T. The broadcast uses for every processor P a variable $down_P$ in which P writes to its sons. In case P is the leader then P repeatedly writes the value of up_P in $down_P$. Otherwise P assigns the value of the $down$ variable of its father to $down_P$.

Using the broadcast topology each processor can repeatedly calculate d, the actual diameter of the network, and get the correct result in every such calculation that follows the first $O(d)$ rounds. Another less memory expensive possibility (i.e. $O(kN)$ bits for a processor) is a protocol that estimates the diameter of the communication graph. It uses the value of the maximal height of the BFS tree of the leader in order to estimate the actual diameter. If the maximal height is h then the diameter of the communication graph is bounded from above by $2h$. The leader has the right h following $O(d)$ rounds of the routing protocol. The leader repeatedly broadcasts the value of h and all the processors use $2h$ as a bound on the diameter of the communication graph.

Once every processor has the right information for d any self-stabilizing protocol PR that assumes the knowledge of d or an upper bound on d (e.g. in [AV-91] d or $2h$ may be used instead of D) could be composed with the topology update protocol or the maximal diameter estimation protocol by a fair protocol composition. If the time complexity of PR is $O(d)$ rounds then the time complexity of the composition is also $O(d)$ rounds.

Acknowledgments: Many thanks to Leslie Lamport for his discussion and to Jennifer L. Welch and anonymous referees for reading a preliminary draft of this paper.

References

[AEV-92] E. Anagnostou, R. El-Yaniv and Vassos Hadzilacos, "Memory Adaptive Self-Stabilizing Protocols", *Proceedings of the 6th International Workshop on Distributed Algorithms, Haifa Israel*, 1992.

[AK+-93] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir and George Varghese, "Time Optimal Self-Stabilizing Synchronization", To appear in STOC-93.

[AV-91] Baruch Awerbuch and George Varghese, "Distributed Program Checking: a Paradigm for Building Self-stabilizing Distributed Protocols", FOCS-91.

[Be-58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87-90,1958.

[BGW-87] G.M. Brown, M.G. Gouda, and C.L. Wu, "Token System that Self-Stabilize", *IEEE Transactions on Computers,* Vol. 38, No. 6, June 1989, pp. 845-852.

[BP-88] J.E. Burns and J. Pachl, "Uniform Self-Stabilizing Rings", *Aegean Workshop On Computing, 1988, Lecture Notes in Computer Science* 319, pp. 391-400.

[Di-74] E.W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control", Communications of the ACM 17,11 (1974), pp. 643-644.

[DIM-90] S. Dolev, A. Israeli and S. Moran, "Self Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity", *Proc. of the Ninth Annual ACM Symposium on Principles of Distributed Computation*, Montreal, August 1990, pp. 103-117.

[DIM-91a] S. Dolev, A. Israeli and S. Moran, "Resource Bounds for Self Stabilizing Message Driven Protocols", *Proc. of the Tenth Annual ACM Symposium on Principles of Distributed Computation*, Montreal, August 1991, pp. 281-294.

[DIM-91b] S. Dolev, A. Israeli and S. Moran, "Uniform Dynamic Self-Stabilizing Leader Election", in *Lecture Notes in Computer Science 579: Distributed Algorithms*, (Proc. of the 5th International Workshop on Distributed Algorithms, Delphi, Greece, October 1991), S. Toueg, P.G. Spirakis and L. Kirousis, Editors, pp. 163-180, Springer Verlag, 1992.

[KP-89] S. Katz and K. J. Perry, "Self-stabilizing extensions for message-passing systems", *Proc. of the Ninth Annual ACM Symposium on Principles of Distributed Computation*, Montreal, August 1990, pp. 91-101.

[Kr-79] H.S.M. Kruijer, "Self-stabilization (in spite of distributed control) in tree-structured systems", Information Processing Letters 8,2 (1979), pp. 91-95.

[La-86] L. Lamport, "The Mutual Exclusion Problem: Part II - Statement and Solutions", *Journal of the Association for Computing Machinery* , Vol. 33 No. 2 (1986), pp. 327-348.

[SG89] J. Spinelli and R.G. Gallager, "Event Driven Topology Broadcast Without Sequence Numbers", *IEEE Transactions on Communication*, Vol. 37, No. 5, (1989) pp. 468-474.

This article was processed using the LaTeX macro package with LLNCS style