Self-Stabilizing Group Communication in Directed Networks*
(Extended Abstract)

Shlomi Dolev and Elad Schiller

Department of Computer Science
Ben-Gurion University of the Negev, Israel
{dolev,schiller}@cs.bgu.ac.il

Abstract. Self-stabilizing group membership service, multicast service, and resource allocation service in directed network are presented. The first group communication algorithm is based on a token circulation over a virtual ring. The second algorithm is based on construction of distributed spanning trees. In addition, a technique that emulates, in a self-stabilizing fashion, any undirected communication network over strongly connected directed network, is presented.

Keywords: Self-Stabilization, Group Communication, and Directed Networks.

1 Introduction

The group communication infrastructure is useful for numerous applications, starting in (video, audio, multimedia) virtual conferences, and including (safety) critical tasks that require exactly once transactions. On line on going systems are prone to unexpected state transition due to transient faults, thus, it is important to design such systems to recover automatically from any possible state.

Previous research towards self-stabilizing group communication services (including membership service, multicast service, and resource allocation services) considered undirected communication networks and ad hoc networks. It turned out that the techniques used for achieving self-stabilizing group communication are different for each such setting. The case of directed networks is important in heterogeneous communication systems that include base-stations, mobile hosts, sensors, servers, satellites, etc. Two-way communication is not always possible in such systems. The focus of this paper is self-stabilizing group communication services for directed networks.

Previous Work: Self-stabilizing group communication for fixed and dynamic undirected networks, and for ad hoc networks (in which servers may move) have

* Partially supported by NSF Award CCR-0098305, IBM faculty award, STRIMM consortium, and Israel ministry of defense. An extended version can be found in [12].

S.-T. Huang and T. Herman (Eds.): SSS 2003, LNCS 2704, pp. 61–76, 2003.

been studied in [11] and [13]. Self-stabilizing algorithms for directed networks have been addressed in [2, 1, 5, 4, 14].

Self-stabilizing mutual exclusion on directed graphs was considered in [2]. The communication graph is assumed to be a strongly connected directed graph that requires (non-distributed) preprocessing (see [16] for such an approach). The authors assume the existence of a distinguish node, a known number n of processors in the system, and of a central daemon [9, 8]. The first algorithm uses a token circulation on a virtual ring. Assuming that the degree of at least two nodes is in $\Theta(n)$, it is shown in [2], that the length of the virtual ring is in $\Omega(n^2)$. The second algorithm uses a spanning tree. The authors assume that the processors know the communication diameter d and that processors have distinct identifiers in the range of $\{0, \ldots, n-1\}$. Roughly speaking, the tree is repeatedly colored by colors in the range $\{0, \ldots, n-1\}$ in a round robin fashion granting the critical section to the processor with the current tree color. Thus, a processor must wait $O(nd)$ time (measured in asynchronous rounds) to reenter the critical section. A self-stabilizing algorithm for leader election and generic tree construction in a strongly connected directed network is presented in [1]. The algorithms of [1] stabilize in $O(n)$ time, where n is the number of processors. (The authors of [1] remark that by a slight change in the algorithm the stabilization time may be reduced, but this change leads to more complex proofs). A self-stabilizing algorithm for routing messages in a strongly connected directed network is presented in [5]. In [5] it is assumed that a distinguish processor exists and that every processor knows the exact number of processors in the system. A randomized self-stabilizing mutual exclusion algorithm in a uniform (directed) networks is presented in [4]. The algorithm uses a virtual ring that is constructed by keeping a pointer in each node and changing it in a round robin fashion. Delaet and Tixeuil presented in [14] a self-stabilizing census algorithm for strongly connected directed networks.

Our Contribution: We introduce the first self-stabilizing algorithms for group communication in directed networks using new building blocks such as membership and multicast services. We do not assume the existence of a distinguish processor, a central daemon, or that the actual number of processors (or the communication graph diameter) is known. Moreover, we do not require preprocessing. We prove that the length of a virtual ring is in $\Theta(n^2)$, even when the degree of every node is in at most three. Simple proofs show that our algorithms stabilize within the order of the system diameter. A transformer algorithm that emulates any undirected network over a strongly connected directed network is presented. In addition, we introduce a resource allocation algorithm for (asynchronous) strongly connected directed networks and (synchronous) weakly connected directed networks.

The rest of the paper is organized as follow. The system settings appear in Section 2. In Section 3, we specify our requirements from a group communication system, and design algorithms for achieving these requirements in Section 4. A general scheme for self-stabilizing emulation, of any undirected communication

network over the directed network, is presented in Section 5. Resource allocation schemes appear in Section 6. Concluding remarks are in Section 7. Proofs are omitted from this extended abstract.

2 The System

The system consists of a set $\mathcal{P}$ of communicating entities, that we call *processors*. A processor is either a physical CPU or a process (a thread). There are $n < N$ processors in the system each processor has a distinct identifier. We represent the system by a communication graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where every node in $\mathcal{V}$ represents a processor and every directed link in $(i, j) \in \mathcal{E}$ represents the possibility of p_i to communicate information to p_j. The device that conveys the information from p_i to p_j can be based on either wire or wireless technology. We model a communication link (i, j) by a *buffer* (or communication register) that stores the last message sent by p_i to p_j, and was not yet received. Thus, there is at most one message in every link and this message is stored in $buffer_{ij}$ with an indication of whether p_j has already received it (one may assume that send operations from p_i to p_j are spread enough to allow a message to arrive before the next message is sent).

The $in\text{-}neighbors_i$ is a set that consists of all the nodes p_j such that the directed link (j, i) is in $\mathcal{E}$. The $out\text{-}neighbors_i$ set is defined analogously. Processors may crash and recover during the execution. However, we concern ourselves with a period of time in which the communication graph is fixed for a long enough period that allows each processor to identify the correct $in\text{-}neighbors_i$ and $out\text{-}neighbors_i$ sets. The processors execute *atomic steps*. An atomic step consists of internal computations followed by a single *communication operation*. A *communication operation* of p_i through the out-going link (i, j) is a message send operation (a write to $buffer_{ij}$). A *communication operation* of p_i through the in-going link (j, i) is a message receive operation (a read from $buffer_{ji}$).

We define a system configuration as a vector of states of all the processors and the content of every buffer. A system *execution* (or *run*) $R = (c_1, a_1, c_2, a_2, \cdots)$ is an alternating sequence of configurations and steps. The system is asynchronous (though we remark that synchronous settings are considered in Section 6). The delay in message delivery (completeness of a write operation) is unbounded but finite. We measure the time complexity by the number of *asynchronous cycles* in an execution. Let R be an execution, and let $\mathcal{S}$ be a strongly connected component of the communication graph. We assume that no processor in $\mathcal{S}$ crashes during R. The first *asynchronous cycle* of $\mathcal{S}$ in R is a minimal prefix of R such that each processor p_i in $\mathcal{S}$ communicates with every neighbor: At least one message m_j is sent by p_i to every neighbor p_j in $out\text{-}neighbors_i$, such that p_j receives m_j during the first asynchronous cycle. The second asynchronous cycle of $\mathcal{S}$ in R is the first asynchronous cycle in the execution R', which starts in the configuration that immediately follows the first asynchronous cycle of $\mathcal{S}$ in R. In a similar manner, we define the rest of the asynchronous cycles. A *fair execution* is an execution with infinite number of asynchronous cycles.

The *task* τ of the system is defined by a set of legal executions *LE*. A configuration c is *safe* with relation to τ, and the system, if every execution that starts from c belongs to *LE*. We require that a self-stabilizing algorithm reaches a safe configuration within a certain number of asynchronous cycles in any execution that starts in an *arbitrary* initial configuration.

3 Group Communication Specifications

Group communication system typically provides various types of multicast services to a specific (dynamic) interest group. A member in the group may specify that a message should be delivered to the rest of the group members with certain reliability requirements and ordering guarantees. The multicast service is then responsible for delivering this message to the application layer of the group members under the defined requirements.

Several groups may coexist simultaneously in the system. However, we assume that no interaction among groups exists. Therefore, we choose a single group index g. A boolean variable $member_i$ is defined to (logically) represents the intention of p_i to be included in g. We use $v_i = \langle vid_i, members_i \rangle$ to represent the view maintained by processor p_i, where vid_i is a unique bounded integer *view identifier*, and $members_i$ is a list of processors indices. Roughly speaking, view identifier is used to distinguish two different incarnations of groups with the same set of members. Our requirements for message delivery are related to the view in which the message has been sent; requiring that every processor that belongs to the view in which the message has been sent and to every following view (if such views exist) will receive the message. For example, assume that message m has been sent by processor p_i when the view was $\langle 22, \{i, j, k\} \rangle$ and later only two more new views $\langle 23, \{i, j\} \rangle$ and $\langle 24, \{i, j, k\} \rangle$ were established, then p_k will not necessarily receive m. p_k can conclude that messages sent in view $\langle 22, \{i, j, k\} \rangle$ may not reach it. On the other hand, if the view identifier of the first (namely, 22) and the last (namely, 24) views were identical p_k could not conclude that some messages may never reach it. Another aspect of the above observation is that these definitions allow us to garbage collect old views (and messages) from the history.

Self-stabilizing algorithms use bounded variables, since a transient fault (or initial state) may corrupt the variable value and cause the variable to have the biggest value at once. Thus, we increment the view-identifier using a modulo operation over a bound V. We assume that all the processors participate periodically in establishing a view. This periodic establishment ensures conflict resolution among view identifiers (if exists) and allow the use of the modulo operation without losing the ordering among views. As we show in the sequel the participating process will rapidly have consistent histories.

Statements of requirements for self-stabilizing group communication appear in [11] and [13]. The first requirement is that in a legal execution every processor that wishes to join (or leave) the group eventually appears (respectively, does not appear) in the list of members $members_i$ of every configuration. In addition, if

processors do not change their membership, then the *view identifier* of the group, vid_i, is eventually fixed.

The set of legal executions LE_{mem} for the group membership task, is associated with $\mathcal{S}$, and includes executions R satisfying Requirements 1 and 2.

Requirement 1: *If the value of* $member_i = true$ *(*$member_i = false$*) is fixed during* R *then there exists a suffix of* R*, in which* p_i *appears (respectively, does not appear) in all the views of group* g *in the connected component* $\mathcal{S}$*.*

Requirement 2: *If the value of* $member_i$ *of every processor* p_i *of group* g *in the connected component* $\mathcal{S}$ *is fixed during* R *then there exists a suffix, in which all the views of group* g *in the connected component* $\mathcal{S}$ *are identical, the views have the same list of members and the* same view identifier*.*

The set of legal executions LE_{mcast} for the group multicast task, is a subset of LE_{mem}, and includes executions R satisfying Requirements 3, 4 and 5.

Requirement 3: *Suppose that two processors* p_i *and* p_j *are members of every view in* R*. If* m *is a message sent by* p_j *during* R*, then* m *is delivered to* p_i*.*

Requirement 4: *Suppose that the messages* m_0 *and* m_1 *are delivered to processors* p_i *and* p_j *during* R*. If* m_0 *is delivered to* p_i *before* m_1 *is delivered to* p_i*, then* m_0 *is delivered to* p_j *before* m_1 *is delivered to* p_j*.*

Requirement 5: *If a processor* p_i *is a member of two successive views of a group,* v_j *and* v_{j+1}*, then all messages sent in the group while* v_j *was the view of the group, have been delivered to* p_i*.*

Let m be a message sent by processor p_i, when $v_i = v$, then v is the *sending view* of m. If a message m is delivered to the application layer of every member in its sending view, then we say that m is *stable*. The system delivers a safety delivery indication (for m) to the application layer when m is stable. We view this indication as a best effort service; we do not require a delivery of a safety delivery indication, for every stable message.

In the sequel, we present self-stabilizing group communication algorithms in directed network that can achieve requirements 1, 2, 3, 4 and 5.

4 Group Communication Algorithms

In this section, we present two approaches for achieving self-stabilizing group communication in directed network. Token circulation (e.g., [6, 13]) is used in the first approach, while the second approach employs a distributed tree structure (e.g., [11]). Both approaches use the self-stabilizing update algorithm in directed network as a building block.

Update in Directed Networks: The update algorithm [7, 10] (for undirected networks) is useful for routing messages, collecting data, and distributing information. We present a version that is suitable for directed networks and can achieve topology update in directed networks. Then we use the resulting algorithm for achieving a variety of self-stabilizing algorithms: Ring construction, group membership, group multicast, β-synchronizer, general network topology emulation and resource allocation.

For the sake of completeness, we now present the basic ideas used by the (undirected) update algorithm of [7, 10]. Each processor p_i maintains a list $\mathcal{U}_i$ of no more than N tuples $\langle id, dis, parent\rangle$. In a legal execution, it holds that $\mathcal{U}_i$ lists the processors in $\mathcal{S}$. For every processor $p_j \in \mathcal{S}$, there is exactly one tuple $\langle j, dis, k\rangle \in \mathcal{U}_i$. The value of dis is the number of edges in a shortest path from p_i to p_j, and p_k is a neighbor of p_i that is in a shortest path to p_j.

Every processor repeatedly sends $\mathcal{U}$ to its neighbors, and accumulates the received lists of its neighbors in $\mathcal{TU}_i$. The content of $\mathcal{TU}_i$ is an input to a procedure that calculates $\mathcal{U}_i$. The value of the dis field of every tuple in $\mathcal{TU}_i$ is incremented by one. Then, the tuple $\langle i, 0, nil\rangle$ is included in $\mathcal{TU}_i$. For every specific id, we select the tuple with the minimal dis value, and remove the rest from $\mathcal{TU}_i$. We remove every tuple $\langle id, dis, parent\rangle$ such that there exists a positive $z < dis$ and there is no tuple with $dis = z$ in $\mathcal{TU}_i$. Finally we assign the value of $\mathcal{TU}_i$ to $\mathcal{U}_i$.

We now slightly change the update algorithm so that it fits directed networks. We extend the tuples of the update algorithm with a fourth field named *in-neighbors*. The field *in-neighbors* in the tuple $\langle i, 0, nil, in\text{-}neighbors\rangle \in \mathcal{U}_i$ consists of the $in\text{-}neighbors_i$ list. Processor p_i repeatedly sends $\mathcal{U}_i$ to every $p_j \in out\text{-}neighbors_i$. The procedure that calculates $\mathcal{U}_i$ from $\mathcal{TU}_i$, is the same for both directed and undirected versions of the update algorithm. The value of the list $in\text{-}neighbors_i$ of every processor propagates to the entire connected component.

The correctness proof starts in the observation that following the first asynchronous cycle the tuples with distance field value 0 are correct, namely for every i there is a single tuple in $\mathcal{U}_i$ with distance 0, the tuple $\langle i, 0, null, in\text{-}neighbors_i\rangle$. Then, in the following cycle the tuples with distance field value 1 are computed using the tuples of the neighbors that have a distance value 0 and therefore are correct. The proof is completed by an induction over the value of the distance fields. The proof shows that within the order of the diameter of the graph, $\mathcal{U}_i$ contains the local topology of every processor p_i, and therefore, the connected component of p_i is known to p_i. The arguments are almost identical to the arguments presented in [8] for the undirected case, see also [14] for a similar proof for the census task (note that the census task, does not notify the processors with the system topology).

Next we describe how the directed version of the update algorithm is used as an underlying layer by several algorithms.

4.1 Token Algorithm:

The pioneering self-stabilizing algorithm of Dijkstra [9] uses a token ring. Token ring circulation was previously used by group communication algorithms

(e.g., [6]). We describe how to construct a ring in a directed network, and the way to use it for a self-stabilizing membership and multicast services.

Token Ring Construction: A virtual ring construction (in a directed network) is defined by a function $next_i(p_j)$, a function from *in-neighbors*$_i$ to *out-neighbors*$_i$. We use the virtual ring definition to forward a token $\mathcal{T}$ (a short message) that arrives to p_i from p_j, to $next_i(p_j)$.

We call the forwarding process of $\mathcal{T}$ a *walk*. A walk defines a ring if, and only if, $\mathcal{T}$ arrives at every processor in the system at least once before returning to the processor in which the walk started in. Moreover, we require that a distinguish processor will be defined in the ring. Every processor p_i may access a predict $distinguish_i$, and there is exactly one processor in the system, such that $distinguish_i = true$.

We now present a straightforward approach for constructing a virtual directed ring (later we present more sophisticate schemes). List the processors according to their identifiers' value, and find a directed path from every two neighboring processors in the list (the graph is strongly connected and hence such a path must exist). Note that every processor can compute the virtual ring using the output of the self-stabilizing update algorithm. The number of nodes in the virtual ring is at most N^2. The distinguish (virtual) processor (represented by) the processor with the maximal identifier according to the update information, is defined to be the first processor in the virtual ring.

The algorithm in [9] is used on top of the virtual ring. Each processor p_i may act as several virtual processors $p_{i1}, p_{i2}, \cdots, p_{il}$, where l is the number of times p_i is visited during the walk. p_i maintains a variable x_{ij} for each p_{ij}. x_{ij} stores an integer value that is no smaller than zero and no larger than $2N^2 + 1$ (there are at most N^2 buffers in the virtual ring and N^2 local read values, we add one more possible value to ensure the existence of a missing value [8]). To define the configuration of the virtual ring we order the values of the x_{kj} variables according to their order of appearance in the (virtual) ring. p_i acts for every virtual node p_{ij}, repeatedly sending the value of every x_{ij} variables to the next node in the virtual ring. The message sent with x_{ij} carries additional values that may be interpreted as the values carried by a token. A token $\mathcal{T}$ *arrives* at the (virtual) distinguish node p_{ij}, if the value of x_{ij} is the same as the value in the arriving message sent by the (virtual) node p_{km} that precedes p_{ij} in the virtual ring. A token $\mathcal{T}$ *arrives* at a (virtual) non-distinguish node p_{ij}, if the value of x_{ij} differs from the value arriving to p_{ij} in the message sent by the (virtual) node p_{km} that precede p_{ij} in the (virtual) ring. In the sequel $\mathcal{T}.y$ refers to the value of the y field in the token $\mathcal{T}$.

Upon the arrival of the token at the distinguish node p_{ij}, p_{ij} increases x_{ij} by one modulo $2N^2 + 1$. Upon a token arrival at a non-distinguish node p_{ij}, p_{ij} assigns the value $\mathcal{T}.x$ to x_{ij}. A node p_{ij} (distinguish or not), repeatedly sends the value of x_{ij} together with the rest of the fields (that may form a token) to the next node in the virtual ring.

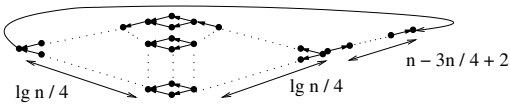

Fig. 1. A lower bound for the length of a ring

Tight Lower Bound for the Length of the Virtual Ring: We now turn to examine the problem of forming a virtual ring in a directed network. An Euler tour over a spanning tree of an undirected network forms a virtual ring of $\Theta(n)$ virtual nodes. We next show that the length of a virtual ring in a strongly connected *directed* network is $\Theta(n^2)$. Our lower bound proof uses only constant degree nodes in the system, as opposed to the assumption of degree in the order of n made in [2].

Assume that $n = 2^i$ for some integer $i > 1$. We construct two full binary trees of depth $d = \lg(n/4)$ one directed from the root to the leaves and the other from the leaves to the root. The number of leaves in each tree is $2^d = n/4$. Then we fuse the leaves of the trees one by one to form the directed *diamond* structure like the one presented in Figure 1. The diamond structure includes $2n/4 - 2 + n/4 = 3n/4 - 2$ processors. In addition, we connect the roots of the (fused) trees by a chain of $n - 3n/4 + 2$ processors.

Since any virtual ring must include all fused processors, and a path between any two fused processors is of length greater than $n/4 + 2$, then the length of the ring is in $\Omega(n^2)$. Since a virtual directed ring can be constructed with at most n^2 processors, we conclude that the length of the ring is in $\Theta(n^2)$.

Optimizing the Length of the Virtual Ring: We now present more sophisticated schemes for token ring configuration in which the length of the ring is optimized. Note that the virtual ring may include exactly n nodes (when the directed network is a directed ring). The problem of finding an optimal length virtual ring in a directed graph is known to be NP-hard [3]. Therefore, we may use an approximation algorithm to find an efficient virtual ring. Self-stabilizing update distributes the information on the communication graph to every processor in the system. Every processor may use a traveling salesman approximation algorithm (see [3]) as a heuristic method to find a near optimal Hamiltonian walk. The "salesman" should visit every city (a processor in $\mathcal{G}$) at least once. The cost of traveling among the cities u and v is ∞, if there is no directed edge from u to v in $\mathcal{G}$, and one otherwise.

We note that both deterministic and randomized approximation algorithms for the traveling salesman can be used. If a deterministic approximation algorithm is executed then the resulting virtual ring is identical in every processor. To use a randomized approximation algorithm we let the distinguish processor calculate the ring and then propagate it to the rest of the processors using the spanning tree rooted by itself (defined by the update algorithm). The distinguish

processor will change the description of the ring that it propagates only when the current propagated ring has a length greater than the last computed ring.

Membership (Token Algorithm): We describe a membership algorithm that satisfies Requirements 1 and 2. Since there is no interaction between any two groups, we consider a specific group g and describe the membership service for g. We use the algorithm in [9] as an underlying algorithm. During the stabilization period of the underlying algorithms, there may be more than one token. When two tokens or more arrive at a single processor, a merge procedure is invoked. The tokens are merged to a single token where the merged token is either empty (see [13]), or is the result of resolving the minimum number of conflicts (see [11]).

We use time to live method (e.g., [15]) to remove stalled information from the token. The token carries a list, $members_g$, of processors that are members in the group g. The token also carries a list of corresponding time to live counters lvs; a counter value, lv_j, for each virtual processor in the virtual ring. The virtual processor p_i assigns the length of the virtual ring to lv_i, whenever a token arrives at p_i. Whenever a processor in the ring receives a token, it decrements the value of each $lv_i \in lvs$. While $lv_i > 0$, we consider p_i as an *active member* in g.

We note that we can order the virtual processors that each processor acts for according to the order of their appearance in the virtual ring. Thus, a processor p_m acting for $p_{m1}, p_{m2}, \cdots, p_{ml}$ will have only one counter in the lvs counters. lv_m (or lv_{m1}) will be set to n whenever the token arrives at p_{m1} and will be decremented by one only when arriving to the first representative of other processors, for example when arriving to p_{l1} that represents p_l. Here we present a version in which each virtual processor has its own counter, and an indication that one virtual processor is not active implies that the acting processor for this virtual processor is not active.

Processor p_i, chooses a new view identifier, whenever it holds a token, and discovers that the set of members in the group has changed. A change in the set of members can occur when a processor p_i voluntarily changes its membership status in group g, or there is an identification that processor p_k is *not* active.

The code of the membership algorithm is presented in Figure 2. Upon token $\mathcal{T}$'s arrival, we decrement every lv counter by one (line 1.1). Line 1.2 stores the current list of members (later used in line 1.6). Lines 1.3 to 1.3.2 remove a processor that has been flagged as not active. Lines 1.4 to 1.4.2 (1.5) adds (respectively, removes) i to (from) the members list upon a request. In the case, there were changes to the members of g, we establish a new view (lines 1.6). Line 1.11 forwards the token to the next neighbor in the virtual ring.

Multicast (Token Algorithm): The multicast algorithm that satisfies Requirements 3, 4 and 5. The *history* is a list of views and messages that should list the views of the system and the messages sent in each view according to the order of the view establishments and message send operations within the views. Whenever the token arrives at a processor p_j, the views and messages

Global constants:
V: upper bound on the number of view identifiers that can be concurrently active
$RingLength$: the number of nodes in the virtual ring
Token $\mathcal{T}$ data structure has fields:
$\mathcal{T}.members_g$: the set of processors in group g
$\mathcal{T}.lv_k$: (where p_k are the processors in $\mathcal{T}.members_g$) counter
for the time to live of p_k
$\mathcal{T}.vid$: identifier for the current view of the group
Local variables of processor p_i:
g_i: boolean indicating whether p_i is a member
v_i: the value of the view for group g recorded at p_i
1. **Upon a token $\mathcal{T}$ arrival from p_k:**
1.1 for-every $\mathcal{T}.lv_k \in \mathcal{T}.lvs$ $\mathcal{T}.lv_k \leftarrow \mathcal{T}.lv_k - 1$
1.2 $members \leftarrow \mathcal{T}.members_g$
1.3 for-every $\mathcal{T}.lv_k \leq 0$
1.3.1 remove k from $\mathcal{T}.members_g$
1.3.2 remove lv_k from $\mathcal{T}.lvs$
1.4 if $g_i = true$ then
1.4.1 add i to $\mathcal{T}.members_g$
1.4.2 add $lv_i = RingLength$ to $\mathcal{T}.lvs$
1.5 else remove i from $\mathcal{T}.members_g$
1.6 if $members \neq \mathcal{T}.members_g$ then $\mathcal{T}.vid \leftarrow \mathcal{T}.vid + 1$ modulo V
($* For\ Multicast *$)
1.11 send $\mathcal{T}$ to $next_i(p_j)$

Fig. 2. Self-stabilizing Membership in a Ring Algorithm, code for p_i

are delivered in a first-in first-out manner (with an appropriate view identifier) to p_j.

A view of a group becomes *old* when a new view has been established to the same group g. An old view $view_o$ is removed from the history, when every processor p_i, member in $view_o$ *and* in the current view, received all the multicast messages of $view_o$ (and also their delivery indications). A message is removed from the history also when the views are not changed following the send operation of the message, but there is an indication that every processor in the view received the message (and their delivery indication).

The code of the multicast algorithm is presented in Figure 3. Line 1.6.1 concatenates the current view with the history of views and messages. Line 1.7 adds the message m that p_i wishes to multicast to the list of messages of $\mathcal{T}$ that are related to the last established view (current view). Line 1.8 delivers new views to the application layer (and mark them as delivered to p_i, thus they are not new to p_i). Line 1.9 deliver new multicast messages (and mark them as delivered to p_i, thus they are not new to p_i). Lines 1.10 to 1.10.1 deliver safe indication for every view and multicast message that have been delivered to all the processors that they should be delivered to.

1.6.1 $\mathcal{T}.history \leftarrow \mathcal{T}.history + \langle \mathcal{T}.vid, members \rangle$
1.7 if p_i wishes to send message m then add m to $\mathcal{T}.messages$ of current view
1.8 for-every non-delivered view $v \in \mathcal{T}.history$ deliver v to the application layer
1.9 for-every non-delivered $m \in \mathcal{T}.messages$ deliver m to the application layer
1.10 for-every safe x that was not reported (x a safe view or message)
1.10.1 deliver report that x is safe

Fig. 3. Self-stabilizing Multicast in a Ring Algorithm, code for p_i

4.2 Tree Algorithm:

The second approach we present is based on a distributed tree structure defined by the update algorithm, and then the self-stabilizing β-synchronizer in directed network. The algorithm presented in this section executes the group communication tasks in the fastest time (in the order of the diameter) while the approach presented in Section 4.1 responds to a group membership request in time that is proportional to n the number of processors (or even to n^2). We note that the tree solution allows several messages that carry the group communication activity to be presented in a configuration, while in the ring solution there is at most one such message (see [13]).

β-Synchronizer: We use the β-synchronizer to coordinate view-updates in the membership service. The undirected β-synchronizer algorithm [8], uses a rooted tree. The processor p_l with the maximal identifier in $\mathcal{U}_i$ is the root of the tree. The tree is repeated colored by a finite set of $colors$.

Here we use two trees to implement the directed version of the β-synchronizer. Namely, we use the distributed tree structures $out\text{-}tree_l$, and $in\text{-}tree_l$. $out\text{-}tree_l$ is used for broadcasting and $in\text{-}tree_l$ for receiving feedback. Roughly speaking, the β-synchronizer makes two alternating phases named "propagation phase" and "feedback phase". Every pair of consecutive (successful) propagation phase and (successful) feedback phase is associated with a color chosen by the root p_l.

The root p_l repeatedly colors $out\text{-}tree_l$ and $in\text{-}tree_l$. Suppose p_f is a leaf processor in $out\text{-}tree_l$, then the color of p_l propagates downwards until it reaches p_f (*propagation phase*). Eventually, after the new color reaches p_f, all p_f's children in $in\text{-}tree_l$, have the same color as the color of p_f. This color is propagated on the path of $in\text{-}tree_l$ that is directed from p_f to p_l (*feedback phase*). When p_l receives a feedback on the new color propagation via $in\text{-}tree_l$, it chooses another color (in a round robin fashion, from a set of $4N$ colors).

Note that (unlike the undirected version of the algorithm) a legal execution might include a configuration in which the children of a processor p_i in $in\text{-}tree_l$ are colored with a new color d, where its parent in $out\text{-}tree_l$ is not colored with color d. This is a consequence of the existence of two different trees; one for propagation, and another for feedback.

Membership (Tree Algorithm): In a legal execution, only the user is privileged to change his/her membership status in a group. Such a change occurs in response to the application request. Here we describe how processor p_i may join (leave) a group g by local setting (respectively, resetting) $member_i$. We use the β-synchronizer to synchronize the changes of views as done in [11].

In a legal execution, processor p_l is responsible for membership updates. During the propagation phase, p_l propagates the view it maintains, v_l, together with a new color. The feedback phase is intended to inform on the completion of the propagation phase, and to gather membership requests from the application.

The values of $member_i$ are accumulated from every processor in $in\text{-}tree_l$ during the feedback phase. Once the feedback phase is completed, the root sends (during the next coloring phase) the received membership information, together with a view identifier. The view identifier changes merely when the set of members changes. Processor p_i accumulates the membership requests in a variable named $request_i$. We define $request_i[i]$ to be an alias to $member_i$. The entry, $request_i[j]$, is reserved for the value of $request_j$, where p_j is a predecessor processor of p_i in $in\text{-}tree_l$. The variable $request_i$ is an array of bits that are associated with every node in the sub-tree of $in\text{-}tree_l$ that is rooted at p_i (the k-th bit in the array is the k-th processor in a pre-order traversal on this sub-tree). The *members* data structure of a view v_i is an array of bits with a structure that is identical to $request_l$ (where p_l is the root of $in\text{-}tree_l$).

The code of the β-synchronizer and the membership algorithm is presented in Figure 4. In line 1.1, the root (of the trees) tests whether the feedback phase has just ended. Line 1.1.1 chooses a new color. Line 1.1.2 establish a new view if the membership group g changes. Lines 1.2 to 1.2.2 are executed by processors other than the root. A message sent by these lines includes colors (for propagation and feedback) and the current view. In lines 2 to 2.3, the propagation phase color and current view are stored. In lines 3 to 3.3, the feedback phase color and membership requests are accumulated. For the sake of simplicity, we assume that $member_i$ is updated just before p_i reports to its parent in $in\text{-}tree_l$ on the completion of the feedback phase.

Multicast (Tree Algorithm): Here we describe a multicast algorithm that delivers messages and feedbacks to the members of the group. For the sake of simplicity, we assume that there is a single message that a processor p_i wishes to send (maybe an aggregation of several data-messages that are stored between consecutive feedback phases) whenever a color is changed in dc_i.

When p_i, wishes to send a multicast message it sends it towards p_l using $in\text{-}tree_l$. Upon the arrival of such a message, p_l uses $out\text{-}tree_l$ to deliver the message to every processor. The change in colors indicates the safe delivery of messages.

We extend the messages of the algorithm presented in Figure 4 with two additional fields dm (down message), and um (up message). Processor p_i stores its multicast message in $um[i]$. The messages that are in um_j (of every child p_j of p_i in $in\text{-}tree_l$) are sent from p_j to p_i, and stored in $um[j]$.

Local variables of processor p_i:
in-parent: the successor processor of p_i ($in\text{-}tree_l$)
in-children: the predecessor processors of p_i ($in\text{-}tree_l$)
out-parent: the predecessor processor of p_i ($out\text{-}tree_l$)
out-children: the successor processors of p_i ($out\text{-}tree_l$)
$uc, dc, uc[]$: two color variables and an array of colors used for synchronization
$req[]$: array that accumulates membership requests ($req[i]$ is $member_i$)
1 **Upon timeout:**
1.1 if $out\text{-}parent = NULL$ and $\forall p_k \in in\text{-}children\ dc = uc[k]$ then
1.1.1 $dc \leftarrow dc + 1 \bmod 4N$
1.1.2 if $v.members \neq req$ then $v \leftarrow \langle Choose(ViewIDs - \{v.id\}), req\rangle$
1.2 if $out\text{-}parent \neq NULL$
1.2.1 if $\forall p_k \in in\text{-}children\ dc = uc[k]$ then $uc \leftarrow dc$
1.2.2 send $\langle uc, req\rangle$ to $p_{in-parent}$
1.3 $\forall p_k \in out\text{-}children$ send $\langle dc, v\rangle$ to p_k
2 **Upon receiving** $\langle dc_j, v_j\rangle$ **from** $p_{out-parent}$**:**
2.2 $dc \leftarrow dc_j$
2.3 $v \leftarrow v_j$
3 **Upon receiving** $\langle uc_j, req_j\rangle$ **from** $p_j \in$ ***in-children***$_i$**:**
3.2 $uc[j] \leftarrow uc_j$
3.3 $req[j] \leftarrow req_j$

Fig. 4. The β-Synchronizer and Membership Algorithm on a Tree, code of p_i

Eventually, p_l receives the messages sent by the processors in the system. The value m of um_l arrives at a processor p_i in the system, along with the next new color nc propagating through *out-tree*. We note that the propagating phase that follows the arrival of um_l can indicate that m was safely delivered.

The code of the multicast algorithm is presented in Figure 5. In lines 1.1.3 the root (of the trees) orders the messages sent by the processors of the $in\text{-}tree_l$ (line 1.2.2), before they are sent on the $out\text{-}tree_l$ (line 1.3). Line 2.1 to 2.1.3 deliver the multicast messages and their safe delivery indication. Line 3.1 stores and accumulates the messages sent on the propagation phase.

5 Emulating a General Topology

In this section we present a self-stabilizing scheme to emulate any undirected network over a directed network. Thus we can compose a self-stabilizing algorithm that assumes undirected communication graph with our emulation scheme. In particular we may combine the self-stabilizing group communication algorithm presented in [11] with the emulation scheme. Note that the convergence time and the space complexity of such a composition is usually greater than the ones of a solution that is build for the specific settings.

Assume that the input for the emulation algorithm is a general topology graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, known to all processors. Let $(i, j) \in \mathcal{E}$ be a link from p_i to p_j

Local variables of processor p_i**:**
$dm, um[]$: a variable and array of messages $(um[i] \equiv um)$
1.1.3 $dm \leftarrow \text{Order}(um)$
...
1.2.2 send $\langle uc, req, um \rangle$ to $p_{in-parent}$
1.3 $\forall p_k \in$ *out-children* send $\langle dc, v, dm \rangle$ to p_k
2 **Upon receiving** $\langle dc_j, v_j, dm_j \rangle$ **from** $p_{out-parent}$**:**
2.1 if $dc \neq dc_j$ then
2.1.1 indicate that dm is safe
2.1.2 $dm \leftarrow dm_j$
2.1.3 deliver dm
3 **Upon receiving** $\langle uc_j, req_j, um_j \rangle$ **from** $p_j \in$ ***in-children***$_i$**:**
3.1 if $uc[j] \neq uc_j$ $um[j] \leftarrow um_j$

Fig. 5. The Multicast Algorithm on a Tree, code of p_i

(such that $p_i \notin$ *in-neighbors*$_j$). We present two different approaches to provide an emulation of the edge (j, i) using the directed network.

Communicating over *in-tree*$_l$ and *out-tree*$_l$: We use the two anti-directed paths between p_i and p_j over *in-tree*$_l$ and *out-tree*$_l$ (these paths may traverse p_l for) emulating an undirected edge (i, j). The delay cost of message delivery (measured by the maximal number of hops that a message traverses) is within the order of the communication graph diameter.

Communicating over the Shortest Path: The update algorithm gathers and distributes the information concerning the communication graph topology to every processor in the system. The two shortest paths that are used to emulate the undirected edge (i, j), can be found by performing BFS twice, once from p_i (until reaching p_j) and once from p_j (until reaching p_i).

In order to ensure eventual delivery of messages, we use a self-stabilizing data link algorithm [8] over the two shortest paths that implement a virtual undirected link. Thus, the time complexity for emulating a message send operation over a link may be proportional to the network diameter, and the space overhead is related to the space required to implement the update algorithm [8].

6 Resource Allocation

We describe the resource allocation for a particular resource, the extension to the case in which there are several independent resources, each accessed in a mutual exclusion fashion, is straightforward.

Strongly Connected Networks: The scheme uses the self-stabilizing β-synchronizer presented in Section 4.2 for collecting requests for the resource

and propagating the winning processor, the processor that can use the resource. Several requests may arrive at p_l the root of *in-tree*$_l$ and *out-tree*$_l$. p_l will store and manage the requests using a local queue *queue*$_l$. The feedback phase collects in a *requests* list (using *in-tree*$_l$) the current requests in the system. p_l removes from *queue*$_l$ processors that did not request the resource during the feedback phase. Then p_l grants the resource to the processor p_i at the head of *queue*$_l$. p_l sends the identifier of the processor (p_i) that is allowed to access the resource to all the processors during the next propagation phase.

Weakly Connected Networks: We propose a self-stabilizing scheme for the case of synchronous systems (see e.g., [8]) in which there exists *single* strongly connected component from which there is a directed path to every other processor. Note that it is possible that a processor p_i may be reached from the strongly connected component but there is no path from p_i to any processor in the strongly connected component.

The processors in the strongly connected component learn the topology of the strongly connected component using the self-stabilizing directed update. Each processor p_i, that has $k \geq 1$ outgoing links that are not part of the strongly connected component, requests the resource on behalf of itself and on the behalf of the processors reached through these k outgoing links. We use the term *mysterious out-going* link for each of these k outgoing links. Once p_i is granted the resource (in a procedure that is identical to the one described for the strongly connected case) p_i uses the resource in a fair fashion, say, use the resource once in every $k+1$ successive times in which the resource is granted to it. In each of the rest of the $k+1$ times, p_i grants the resource to (the subsystem connected through) a (distinct) processor (chosen in a round robin fashion) connected via one of its mysterious edges. p_i always release the resource following B time units knowing that any (processor in such) subsystem that received the grant (lease) for the resource must release it within B time units.

7 Concluding Remarks

This paper presents schemes for achieving self-stabilizing group communication services in directed networks. We view our study as an important building block in understanding the possible and impossible middleware services in a faulty environment. Our lower bounds and techniques can be used for achieving other tasks in a directed network, in fact we show that we can simulate *any* undirected network over a strongly connected directed network in a self-stabilizing fashion.

References

[1] Y. Afek and A. Bremler, "Self-Stabilizing Unidirectional Network Algorithms by Power-Supply," *Chicago Journal of Theoretical Computer Science*, Vol. 1998, No. 3, December 1998. 62

[2] D. Alstein, J. H. Hoepman, B. E. Olivier, P. I. A. van der Put, "Self-Stabilizing Mutual Exclusion on Directed Graphs," In *Computing Science in the Netherlands (Utrecht, 1994)*, Stichting Mathematisch Centrum, Amsterdam, 1994, pp. 42–53. 62, 68

[3] J. Bang-Jensen and G. Gutin, "Digraphs: Theory, Algorithms and Applications," Springer Monographs in Mathematics, Springer-Verlag, London, 2000. 68

[4] J. Beauquier, M. Gradinariu, C. Johnen, J. Durand-Lose, "Token Based Self-Stabilizing uniform Algorithms," *Journal of Parallel and Distributed Computing (JPDC),* 62(5):899–921, May 2002. 62

[5] J. A. Cobb and M. G. Gouda, "Stabilization of Routing in Directed Networks," *Proc. 5th Workshop on Self-Stabilization Systems*, pp. 51–66. 62

[6] F. Cristian and F. Schmuck, "Agreeing on Processor Group Membership in Asynchronous Distributed Systems," Technical Report CSE95-428, Department of Computer Science, University of California San Diego, 1995. 65, 67

[7] S. Dolev, "Self-Stabilizing Routing and Related Protocols," *Journal of Parallel and Distributed Computing*, 42:122–127, May 1997. 66

[8] S. Dolev, *Self-Stabilization*, MIT Press, 2000. 62, 66, 67, 71, 74, 75

[9] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communication of the ACM*, 17:643–644, 1974. 62, 66, 67, 69

[10] S. Dolev and T. Herman, Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997. 66

[11] S. Dolev and E. Schiller, "Communication Adaptive Self-stabilizing Group Membership Service," *5th Workshop on Self-Stabilizing Systems*, October, 2001. Also, Technical Report `#00-02` Department of Computer Science Ben-Gurion University, Beer-Sheva, Israel, 2000. 62, 64, 65, 69, 72, 73

[12] S. Dolev and E. Schiller, "Self-Stabilizing Group Communication in Directed Networks," Technical Report `#10-03` Department of Computer Science Ben-Gurion University, Beer-Sheva, Israel, 2003. 61

[13] S. Dolev, E. Schiller, and J. L. Welch, "Random Walk for Self-Stabilizing Group Communication in Ad-Hoc Networks," *21st Symposium on Reliable Distributed Systems,* IEEE Computer Society Press, pp. 70–79, 2002. 62, 64, 65, 69, 71

[14] S. Delaet and S. Tixeuil, "Tolerating transient and intermittent failures," *Journal of Parallel and Distributed Computing,* 62(5):961–981, 2002. 62, 66

[15] A. S. Tanenbaum, *Computer Networks*, Prentice Hall, 1996. 69

[16] M. Tchunte, "Sur l'auto-stabilisation dans un reseau d'ordinateurs," *RAIRO Informatique Theoretique*, 15:47–66, 1981. 62