Random Walk for Self-Stabilizing Group Communication in Ad Hoc Networks

Shlomi Dolev, *Senior Member*, *IEEE*, Elad Schiller, and Jennifer L. Welch

Abstract—We introduce a self-stabilizing group communication system for ad hoc networks. The system design is based on a mobile agent, collecting and distributing information, during a random walk. Three possible settings for modeling the location of the mobile nodes (processors) in the ad hoc network are presented: slow location change, complete random change, and neighbors with probability. The group membership algorithm is based on a mobile agent collecting and distributing information. The new techniques support group membership and multicast, and also support resource allocation.

Index Terms—Ad hoc networks, group communication, self-stabilization, random walk.

1 Introduction

One of the exciting and fast-moving trends in computing is ad hoc communication networks. Recent developments in wireless networking are making mobile computing a viable technology [11]. *Ad hoc networks* [22], [24] do not use preexisting infrastructure (such as base stations or telephone lines), but instead rely solely on wireless links between mobile computers, resulting in "ad hoc" network connectivity topologies.

With the spread of wireless distributed systems, it is imperative to find ways to simplify their programming. One approach, which has already been applied successfully already to "traditional" wired distributed systems, is to provide communication primitives that hide lower-level complications that arise due to (partial) failures and asynchrony of distributed systems. Higher-level applications can then be built on top of these communication primitives. Supporting higher-level applications is the goal of *group communication* services.

The key features of a group communication facility are 1) indicating to each node of the distributed system which "group" it belongs to—that is, with which other nodes it can currently communicate,[1] and 2) letting nodes within a group communicate with each other in an ordered and reliable manner. The first feature is called a *group membership* facility, while the second feature consists of various kinds of broadcasts and multicasts.

1. The group can either consist of all or a subset of the nodes that are physically reachable, thus determined by the network topology. The group may be also defined by a common specific application in which the set of nodes that are members in the group, are interested. In the latter case, the group members may not be within the same connected component all the time.

• *S. Dolev and E. Schiller are with the Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel. E-mail: {dolev, schiller}@cs.bgu.ac.il.*
• *J.L. Welch is with the Department of Computer Science, Texas A&M University, College Station, TX 77843-3112. E-mail: welch@cs.tamu.edu.*

Manuscript received 23 June 2004; revised 25 Mar. 2005; accepted 25 Mar. 2005; published online 16 May. 2006.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number TMC-0204-0604.

To gain some intuition, one may imagine a user entering a highly populated area and receiving information on his/her computing device through an ad hoc connection. This may be carried out, say, by a single virtual token that is randomly transferred from one user to the other, where users may add information to the token, and old information is eventually erased from the token.

Fault tolerance is very important in distributed systems that may experience crashes of processors (mobile nodes), failure of communication links, and unexpected noise in message transmission. One kind of processor failure considered in previous work is when the processor crashes and later recovers in a state indicating that it has just recovered from a crash; usually, it is assumed that the processor has access to stable storage which survived the crash. Most previous work on group communication has assumed processor crashes.

Many fault-tolerant algorithms do not consider the case of faults that cause a temporary violation of the failure assumptions made by the algorithm designer. For example, most of the algorithms that are designed to cope with Byzantine faults do not recover if more than one-third of the processors temporarily experience a fault and then continue to execute their program starting from the state following the fault. *Self-stabilizing* [12], [13] algorithms cope with the occurrence of temporary faults in an elegant way. A self-stabilizing algorithm can be started in *any* global state, which might occur due to the occurrence of an arbitrary combination of failures. From that arbitrary starting point, it must ensure that the task of the algorithm is accomplished, provided that the system obeys the designer's assumptions for a sufficiently long period. An algorithm is shown to be self-stabilizing by showing that, starting in an arbitrary state and assuming no further failures occur, eventually the algorithm will eventually solve the problem of interest.

Since a group communication layer is "middleware" for a distributed system, it is designed to execute forever, like an operating system. Thus, it is highly unlikely that it will never experience a transient failure, especially in highly dynamic, wireless, mobile networks.

Unlike prior work on self-stabilizing group communication [14], we focus here on algorithms that fit the special characteristics of ad hoc networks. Mobile communication networks, by definition, experience movement of some (or all) of the computing entities. In such a dynamic environment, group communication services must continuously follow the changing locations of the computers in the group; thus, geographic location is a new parameter for problem solutions. As nodes change location relative to each other, connections between nodes can go up and down at a much higher rate than that experienced in so-called "dynamic" (wired) networks subject to link failures and repairs.

Several algorithms for ad hoc networks use flooding in order to reach every mobile node (note that we use the terms *mobile node* and *processor* to denote the mobile hardware communicating computing devices that jointly form the ad hoc network). This approach results in heavy traffic that may use up the limited energy of the mobile processors. In particular, in order to support self-stabilizing group communication services, a large number of flooding multicast messages may arrive *simultaneously* at a single mobile processor. The receiving mobile processor may not be able to process these arriving messages. Here, we take a totally different approach, where one *mobile agent* becomes responsible for broadcasting. We note that the suggested mobile agent serves all the processors (mobile nodes) in the group, as opposed to the common used mobile agent that serves a particular processor (the processor that created the mobile agent).

Another approach used for coordinating the operation of ad hoc networks is to construct and maintain a distributed structure such as a spanning directed acyclic graph (e.g., TORA [30]). This approach can be too optimistic when changes are very frequent; for example, the TORA spanning directed acyclic graph might never be up to date.

In our work, we do not assume that processors need to change their location according to some specific pattern, as suggested in [20], or that there is a set of support hosts [6] that assist in transferring information using random walks. In [6], it is assumed that the support hosts move faster than other hosts and perform a random walk on a regular spanning tree (a tree with a fixed degree for nodes), whose vertices represent partitions of the space in which the hosts move. The ideas of snakes and runners are presented; both are based on forcing the processors in the support set to move in a specific way. In contrast, we analyze particular cases in which the communication graph of the system changes dynamically. We do not rely on the movements of the processors; instead, processors send an *agent* that traverses the (dynamic) graph in a random walk fashion.

Finally, we note that mobile agents in the context of self-stabilization were first studied in [23], [19] and then in [4], [21]. To the best of our knowledge, previous works have considered fixed communication graphs and have not addressed group communication services.

Our Contribution. We present a new approach for achieving a self-stabilizing group communication service in ad hoc networks. Our approach is based on a random walk of an agent. Therefore, we do not have to maintain a distributed structure, such as a directed acyclic graph, in a self-stabilizing manner.

We consider the following new analytical approaches for ad hoc networks to be as an important contribution of our work. Three possible settings for modeling the location of the processors in the ad hoc network are presented: slow location change, complete random change, and neighbors with probability. The group membership algorithm is based on collecting and distributing information by a mobile agent. We detail the way the new techniques support group membership and multicast, and offer an example application—resource allocation.

The rest of the paper is organized as follows: The system settings appear in Section 2. In Section 3, we discuss and analyze random walks under different assumptions about the mobility pattern exhibited by an ad hoc network. The group membership algorithm is presented in Section 4 and the group multicast algorithm in Section 5. The resource allocation algorithm is presented in Section 6. Concluding remarks are in Section 7.

2 The System Settings

In this section, we detail the settings of the ad hoc communication system. An ad hoc communication system does not assume the existence of a fixed communication infrastructure.

The system consists of communicating entities, which we call *processors*. We denote the set of processors by $\mathcal{P}$, where $|\mathcal{P}| = n \leq N$. N is an upper bound on n, the actual number of processors in the system, and (unlike n) is known to the processors. We assume that n is fixed during the period of interest. In addition, we assume that every processor has a unique identifier.

Every processor p_i executes a program that is a sequence of *steps*. For ease of description, we assume the interleaving model, where steps are executed atomically, a single step at any given time. A step of p_i includes the execution of a sequence of *statements*. In addition, p_i may receive and send (from/to a neighboring processor) a special message, called an *agent*, as the first and the last operation of a step, respectively. Note that it is possible that two (or more) consecutive steps of a certain processor may involve receiving agents. An *agent* is a program coupled with a program state (the program state is also called *briefcase*). The description in terms of mobile agents is more convenient than a description in terms of tokens (although the underlying implementation of both agents and tokens using message-passing is alike.) A token does not carry a program to be executed. The use of agents allows us to change the algorithm on-the-fly without reprogramming all the ad hoc units. The agent program may be repeatedly copied from a reliable and updated source, such as a fixed base station. Whenever the agent arrives at such a station, the new program is replaced with the agent's current program. In the sequel, for example, we present several schemes for achieving group communication in an ad hoc network. It is possible that an outside observer will assist in choosing the current scheme (from a set of possible schemes, or even impose a new scheme fitting his particular system settings) by changing the program carried by the agent. The infrastructure used by the agent is relatively simple; in fact, one may set it up using existing primitives that support weak mobility, such as Ajanta (see [34]).

Processors use the agents to communicate with each other. The program of the agent may have permission to read and write variables of the processors. In this way, processors and agents change one another's state. Agents that arrive at a processor p_i are stored in a set $\mathcal{A}_i$. We assume that $|\mathcal{A}_i| \leq (\Delta + 1)$, where Δ is the maximal possible degree of a node in the graph. Thus, the amount of space required for $\mathcal{A}_i$ is bounded.

Whenever a_j is in $\mathcal{A}_i$, we say that a_j *visits* p_i and p_i *hosts* a_j. p_i executes steps in its own program and steps of an agent from $\mathcal{A}_i$. During the execution of a step of the program of p_i, p_i may receive new agents and then access and modify $\mathcal{A}_i$. For example, p_i may recognize that two agents should "collide" and merge these agents into a single agent in $\mathcal{A}_i$.

When p_i executes a step of a_j it changes the state of a_j to be the state of the program of a_j following this step execution. Then, p_i sends a_j to a neighboring processor p_{next}. The atomic step between two configurations that changes the hosting processor of agent a is called a *move* of agent a. We note that the terms "agent move" and "agent (atomic) step" are different. In the sequel, we choose p_{next} randomly among the neighbors of p_i and, hence, perform a random walk.

The *state* s_i of a processor p_i consists of the value of all the variables of the processor, including the value of its program counter. Every execution of a step in the algorithm changes the state of a processor (in particular it may change the value of $\mathcal{A}_i$). A full description of the state of an ad hoc system at a particular time is a vector $c = (s_1, s_2, \cdots, s_n, G(\mathcal{V}, \mathcal{E}))$ of the states of the processors and the topology of the current communication graph $G(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of processors with their coordinates in the plane, and $\mathcal{E}$ is the set of edges implied by the location of the processors and the common (or individual) communication radius rd of the processors. In other words, a node $p_i \in \mathcal{V}$ represents a processor with its coordinates in the plane, and an edge $(p_i, p_j) \in \mathcal{E}$ represents the fact that p_i and p_j can communicate with each other. Note that we do not assume that processors are aware of their location. We assume that p_i executes an algorithm that discovers the $neighbor\ s_i$ set, such that the geographical distance between p_i and $p_j \in neighbor\ s_i$ is no more than some fixed value rd. $neighbor\ s_i$ is symmetric; that is, $p_j \in neighbor\ s_i$ implies $p_i \in neighbor\ s_j$. The term *system configuration* is used for $c = (s_1, s_2, \cdots, s_n, G(\mathcal{V}, \mathcal{E}))$. Note that we assume that messages (agents) in transit are part of the state (input buffers) of the receiving processors and, therefore, the vector of processor states and the current communication graph fully describes the system state.

We define an *execution* $R = c_0, r_0, c_1, r_1, \ldots$ as an alternating sequence of system configurations c_i and steps r_i, such that each configuration c_{i+1} (except the initial configuration c_0) is obtained from the previous configuration c_i by the execution of the step r_i. Note that r_i is either a step of the program of a processor p or an agent activation by p. In the latter case, an agent may be sent to a processor that is a neighbor of p (where neighbors are defined by the communication graph in c_i). In addition, r_i may be a change in the communication graph due to a movement of p_i. Thus, the only components that can be changed due to the execution of r_i are the state of p, the state of a neighbor of p (p_{next}), and the communication graph $G(\mathcal{V}, \mathcal{E})$. An execution is *fair* if every processor executes a step infinitely often.

In this work, we use random walks for broadcasting information. Thus, we consider (fair) executions in which the random walk succeeds in arriving at all nodes in the system. We define a *nice execution* to be a fair execution in which: 1) there exists a single agent and 2) the single agent arrives at every processor in, at most, every M consecutive agent moves, where M is a constant that depends on n.

In the sequel (in Section 3), we compute the probability of eventually having a nice execution in several common cases. The probability is calculated by assuming that we start with an arbitrary configuration and with an arbitrary number of agents. Then, we prove that every nice execution must eventually solve the task, as explained below.

One key issue that supports nice executions is the radius of the transmission. Note that a large radius essentially results in a complete graph (in which we will show there is high probability for nice executions). Another mechanism, described in the sequel, is the time-out mechanism, which enables the creation of an agent when no agent exists.

The *task* of an ad hoc system is defined by a set of executions *LE*, called *legal* executions. A configuration c is *safe* with regard to a task and the ad hoc system if every nice execution that starts from c belongs to *LE*. We say that the algorithm satisfies its task when its execution is in LE.

We require that a self-stabilizing ad hoc system will satisfy the following conditions: 1) Starting from any arbitrary configuration, eventually the execution becomes nice (for a long enough period) with some positive probability. 2) Then, we require absolutely that every nice execution eventually reaches a safe configuration and, thus, satisfies the task.

A self-stabilizing ad hoc system recovers from transient faults that disturb its behavior for a limited period of time. The correctness of a self-stabilizing ad hoc system is demonstrated by considering every nice execution that follows the last transient fault (note that topology changes are not considered transient faults). The system should exhibit the desired behavior for an infinitely long period after a finite convergence period.

The time complexity of an asynchronous, self-stabilizing, distributed algorithm is measured by asynchronous cycles in a nice execution. Note that every processor executes a step infinitely often in every (infinite) nice execution. The first *asynchronous cycle* of a nice execution R is the shortest prefix, R', of R in which every processor executes at least one step. The second *asynchronous cycle* of R is the first *asynchronous cycle* of R'' where $R = R'R''$. The following asynchronous cycles are defined analogously.

3 Random Walks of Agents

The dynamic nature of ad hoc networks makes it difficult to collect information concerning the current topology of the system. An attempt to collect such information will often result in out-of-date information. Thus, we propose to use random walks (see, e.g., [1], [27]) as the main tool for transferring information.

We now describe a random walk of an agent over a dynamic graph. A processor, p, that is about to send an agent randomly chooses a processor, p_{next}, from among the processors with which it can directly communicate. Then, p sends the agent to p_{next}.

The above simple random walk procedure is used for covering the graph (broadcasting). We define the *expected cover time* of a graph to be the expected number of moves required in order for a single agent to visit every processor at least once. For calculating the expected cover time, we choose a starting point that results in the maximal value for the cover time.[2] We assume that a processor p that holds an agent sends the agent to p_{next} within a constant number of its own steps. Thus, the expected cover time can be described in terms of asynchronous cycles instead of agent moves.

Some of our algorithms assume the existence of a single agent in the system. A self-stabilizing system can be started in an arbitrary configuration where no agent exists, or several agents exist. We use a time-out mechanism in order to address the first situation. When a processor p_i does not receive an agent for a predefined period of time, p_i produces an agent. Agent collisions are used to make sure that a single agent survives. We assume that all the agents move from one processor to a neighboring processor in a single asynchronous cycle. The *expected meeting time* is an upper bound on the expected number of asynchronous cycles until a single agent exists. Note that the expected meeting time is a function of the number of agents in the first configuration of the execution. In the sequel, we show that the complexity of the expected meeting time and the expected cover time in the graphs that we consider is the same.

We next show that it is impossible to ensure that the agent visits every processor in the system when the changes in the communication graph are arbitrary (and controlled by an adversary). We then present common cases in which the random walk succeeds in visiting all the processors. In the latter cases, the resulting executions are nice executions. (Note that Sections 4 through 6 build group communication services for the set of nice executions.)

3.1 Impossibility Result

In this section, we present a simple impossibility result that formally demonstrates that, even when the system is always connected (i.e., there is no permanent cut in the system), the ad hoc nature of the system can prevent an agent from visiting a particular processor.

Suppose that processors p_1, p_2, and p_3 are connected in a ring topology. Assume that p_2 frequently moves toward p_1, loses connection with p_3, and then moves back to p_3, reestablishing the connection with p_3 and losing the connection with p_1. Note that the communication graph is always connected and forms a chain of processors—either p_3, p_1, p_2 (when the connection between p_3 and p_2 is not active), or p_2, p_3, p_1 (when the connection between p_1 and p_2 is not active). Assuming the above topology changes, there is an execution in which the agent never visits p_2. In particular, the agent visits p_1 whenever p_2 is not connected to p_1 and visits p_3 whenever p_2 is not connected to p_3.

2. There exist graphs for which the expected cover time differs for different starting points.

3.2 Viable Communication Graph

Here, we consider the case where an agent infinitely often covers the system. We say that the link from a processor p_i to p_j is *viable* in an execution R if, and only if, an agent traverses the link from p_i to p_j infinitely often. We define $\mathcal{T} = p_{i_1}, \cdots, p_{i_l}$ to be a *viable path* from p_{i_1} to p_{i_l} if the links between p_{i_j} and $p_{i_{j+1}}$ are viable (where $1 \leq j \leq l-1$).

We note that, unless there are two viable paths—one from p_i to p_j, and one from p_j to p_i—there is eventually a permanent cut in the communication graph, such that p_i is in one portion of the graph and p_j is in another portion of the graph. Note that, in the example used for the impossibility result in Section 3.1, there is no viable path between p_1 and p_2.

In order to implement a group communication service, it is not sufficient to have a viable path between every pair of processors. Processors need to make sure that the agent has covered the graph before concluding that the membership has changed. Thus, we restrict our discussion to cases in which there is an expected upper bound for the number of agent moves that are required for covering the graph. Note that the expected upper bound for the number of agent moves can be either given or estimated during the execution.

We now turn to describe several cases in which the agent does visit all the processors in the system. First, we consider the case in which the location changes of the mobile hosts are slow.

3.3 Fixed Communication Graph

The value of the communication radius rd can influence the frequency of changes in the communication graph $G(\mathcal{V}, \mathcal{E})$. At one extreme, rd is big enough to always reach every other processor. In this case, the communication graph is always a fully connected graph. If the communication radius is close enough to the value that is required to reach every processor, however, then only very few changes in the communication graph occur.

Another consideration is the speed of processors with relation to the speed of an agent. We may assume a fixed communication graph when the agent is much faster than the processors. This is the case when the time between two changes in the communication graph is larger than the expected time required for the agent to perform a random walk that covers the graph. The above motivates us to also consider the case of a fixed communication graph.

We also note that the graph is fixed for an agent when the agent's traversal does not visit a node more than once during a certain traversal that covers the graph. In other words, changes in the unvisited portions of the graph do not influence the assumption that the graph is fixed. Similarly, changes in portions of the graph that were visited and are not visited again are also allowed.

For completeness, we point out that the expected meeting time and the expected cover time of a fixed graph have been well studied.

In [9], it is shown that, within $O(n^3)$ agent moves, there is a single agent a. In [15], it is shown that the expected cover time is $O(n^3)$ agent moves.

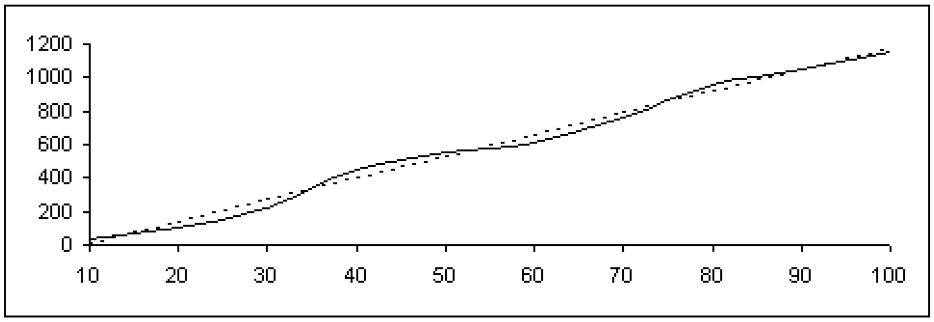


Fig. 1. Average cover time. The number of processors is presented on the horizontal axis. The number of agent moves is presented on the vertical axis. The solid line depicts the average number of moves it takes the agent to cover the system.

3.4 Random Changes in the Graph

Here, we assume the other extreme, where the graph is always connected, but can be totally changed between two successive moves of the agent. We explain that a random walk on a very dynamic communication graph is essentially a random walk on a complete graph. The choice of movement to a neighbor can be viewed as a random choice of the current neighbors and then a random choice of a neighbor from the neighbors set.

The expected cover time for a complete graph is $O(n \log n)$ [16]. In the following lemma, we show that, within expected $O(n \log n)$ agent moves, there is a single agent a. Thus, the expected meeting time of n agents is $O(n \log n)$ agent moves.

Lemma 1. *Let R be a fair execution, such that there are $k > 1$ agents in the first configuration of R, and no other agent is produced in R. Then, within expected $O(n \log n)$ agent moves, there is a single agent in the system.*

Proof. Without loss of generality, we may assume that $k \leq n$, because agents in the same processor collide to become a single agent.

Suppose that a is the first agent to move in R. Then, the probability that a chooses to move to a processor p_i that hosts another agent is $(k-1)/(n-1)$. Therefore, $(n-1)/(k-1)$ is the expected number of agent moves in R before we have a configuration with $k-1$ agents. Since for every $1 < k \leq n$ we have $(k-1)/(n-1) > 0$, the expected number of agent moves it takes for k agents to collide to become one agent is $\Sigma_{i=k-1}^{i=1}(n-1)/i$, which is $O(n \log n)$. □

3.5 Neighborhood Probability

Here, we consider the case in which, for each i, there is a fixed set $\mathcal{N}_i$ of processors, such that each processor in $\mathcal{N}_i$ has a probability $1/|\mathcal{N}_i|$ of being a neighbor of p_i when p_i chooses p_{next}. This case corresponds to situations where processors are always close to their "home location" and, therefore, have a fixed set of neighbors. For example, perhaps two of the ad hoc mobile nodes, a and b, are usually active in one neighborhood and are hardly ever active in another neighborhood where c and d are usually present. Therefore, a and b are more likely than, say, a and c to become neighbors.

An argument similar to the one used in the previous case can be used to prove a reduction to a fixed graph in which p_i is a neighbor of the processors in $\mathcal{N}_i$.

Note that it is possible that the probability for each processor to be in $\mathcal{N}_i$ when p_i chooses p_{next} may not be the same. For instance, these probabilities can be $q_1, q_2, \ldots, q_l$ ($q_j > 0$) for the processors $p_1, p_2, \cdots, p_l$, respectively. In this case, the agent can choose p_j to be p_{next} with probability $(1/q_j)/(\Sigma_{k=1}^{l}(1/q_l))$. Moreover, p_i may repeatedly collect data concerning the neighborhood relation in order to estimate q_j online. In particular, p_i may count during a certain time period the number of times each neighbor has been connected to it and use these numbers to estimate q_j.

The above is in fact a reduction to the case of an arbitrary fixed graph in which p_i is a neighbor of the nodes in N_i. Thus, the expected meeting and expected cover times are $O(n^3)$.

3.6 Network Simulator

We have used the ns-2 [29] network simulator to estimate empirically the expected meeting time and the expected cover time in a situation that is difficult to model analytically; namely, when the nodes move randomly. The results for this case were quite good with observed average times that are even smaller than in the case of a complete graph.

In our simulation, we use normalized units for 1) geographic distances and broadcast distances, 2) density of processors in a geographic area, and 3) time units. Processor movement is simulated by allowing every processor to perform a random walk. In every time unit, every processor executes a step moving, at most, one distance unit. A processor that holds an agent may send the agent to another processor that is, at most, two distance units away. The (average) density of the processors per geographic area of one square distance unit is one.

We simulated agent movement with different numbers of processors $n = 10, 20, \ldots, 100$. The geographic dimensions of the area were set to preserve the processors' density. Fig. 1 depicts the number of agent moves it takes to visit every mobile ad hoc processor in the simulated system. On the horizontal axis, we have the number of processors (i.e., n), and on the vertical axis, we have the number of agent moves. The solid line depicts the average number of moves it takes the agent to cover the system. The dotted line depicts a linear approximation of the moves.

The results show that the average cover time under our choice of settings is approximately a linear function of the number of processors (see [2] for more details).

4 Membership Service by Random Walks

To state the requirements for the self-stabilizing group membership service, we first define the view identifier vid of a group. Every processor p_i in the system has a Boolean flag g_i that indicates whether it wishes to be a member of group g. Group membership can change during the execution, as processors may join and leave a group. Changes in the set of members cause the establishment of a new *view* for the group. A view of a group g is a list of the members of g, denoted $member\ s_g$, which never exceeds N in size, and a view identifier vid. We assume that the agent has a variable in which the vid is stored. We note that a new vid is chosen by incrementing the previous vid modulo a bounded number of identifiers, $\mathcal{I}$, that ensures an ordering between the existing views in the system. The bound on $\mathcal{I}$ is a system parameter (which can be set by the system administrator) that depends on the expected time in between two instances in which *all* the processors are connected and covered by an agent; such an instance may be used to resolve view identifier conflicts, and record a uniform vid value for all the processors. Once the view identifiers are the same, the ordering of the next vids is well-defined as long as the next instance in which all processors are connected takes place before the vid reaches the recorded value once more.

4.1 Group Membership Requirements

Requirement 1. *For every nice execution, R, and every $p_i \in \mathcal{P}$, if g_i has a fixed value during R, then R has a suffix such that p_i appears in $members_g$ if, and only if, $g_i = true$.*

Requirement 2. *Every nice execution in which all the g_i variables have fixed values has a suffix in which vid is not changed.*

Since there is no interaction between groups, we consider a specific group g and describe the membership service for g.

We use an idea that is similar to the time-to-live technique (see, e.g., [33]). The agent carries a list, $member\ s_g$, of members in the group g and a list of corresponding counters lvs—a counter value, lv_i, for each $p_i \in member\ s_g$. Whenever an agent visits p_i, p_i assigns a predefined value ttl_i to lv_i. The value of ttl_i is a function of the (expected) number of agent moves required to cover the communication graph; this value can be changed by p_i when the estimated number of nodes is adjusted (see below). The value of each $lv_i \in lvs$ is decremented by 1 whenever the agent is received by a processor. We say that p_i is an *active member* in g when the value of $lv_i > 0$.

In the sequel, we describe our method for the cases in which the expected cover time is $O(N^3)$. This expected cover time fits the fixed graph case and is also the maximal value among the mobility pattern we have described. Thus, this value should be used if the system administrator does not have a better estimate for the system mobility pattern. Cases in which a better estimate exists for the expected cover time are handled analogously. We use kN^3, for some $k > 1$, as an upper bound for both ttl_i and lv_i, where k is a security parameter on the expected cover time. The bigger k is, the more likely that a node will not be falsely assumed to have dropped out because of an unfortunately long random walk; on the other hand, new views may not be established in a timely fashion when processors are disconnected. In the context of a self-stabilizing algorithm, we have to bound the value of the variables. We note that the values of ttl_i and lv_i cannot exceed kN^3.

In the previous section, we assumed the existence of at least one agent. Since we are interested in a self-stabilizing membership algorithm, we now present techniques that ensure the existence of a single agent. There are two cases to consider:

No agent exists in the system. Here, we use timers rather than an asynchronous distributed algorithm for detecting the fact that no agent exists. The reason is that our approach can be applied to ad hoc networks with changes that are too frequent for performing an asynchronous distributed algorithm to detect the above situation.

The first simple approach uses an upper bound on the expected cover time of the agent (which is a function of N). A processor creates an agent if it does not receive an agent for a time that is proportional to executing $O(CT)$ agent moves, where CT is the expected cover time for the system.

It is possible that, due to a partition, the system will consist of $n << N$ processors. We would like to avoid a waiting time of $O(N^3)$ agent moves, before detecting that an agent does not exist in the system. We describe a technique that enables the processors to create a new agent after a waiting time that is proportional to $O(n^3)$ agent moves. We note that the stabilization time of a self-stabilizing algorithm that has to ensure the existence of an agent and in which processor p_i may wait for T moves, is at least T moves. The reason is that the system can be started in a configuration in which no agent is in the system, and all processors are in a state in which they start to wait for T moves.

In the sequel, we show how the processors can learn the estimated number of processors. We propose that a processor p_i will send a special agent, called *scouter*, following a predefined (relatively short) time period in which no scouter visited p_i. A scouter will update the processors regarding the estimated number of processors once it discovers (with high probability) that it has covered the entire system. A scouter is a "light weight" agent that does not execute the group communication algorithms. The briefcase of a scouter contains a list of the processors it visited. A collision of scouters at a processor eliminates all scouters except one. More details are given in Section 4.1.

Several agents in the system. We use the fact that agents collide during their random walks to ensure the reduction of the number of agents. A collision occurs when the agents are listed together in the set $\mathcal{A}_i$ of a processor p_i. For simplicity, we set the value of $member\ s_g$ of the new agent to include only the identifier of the processor, p_i, that created the agent and the value of lv_i.

Whenever a processor p_i that holds an agent discovers that the set of members in the group has changed, it chooses a new view identifier. A change in the set of members can be due to the fact that a processor voluntarily changes its membership status in group g, or an identification of a connection loss with a processor p_k (when $lv_k \leq 0$).

The formal description of the algorithm appears in Fig. 2. Upon agent a's arrival, we decrement every lv counter by

```
Global Constants:
$ttl_i$: time to live, $k$ times the expected
  cover time
$\mathcal{I}$: upper bound on the number of view
  identifiers that can be concurrently
  active
Agent Data Structure has Fields:
$a.members_g$: set of processors in group $g$
$a.lv_j$: (where $p_j$ are the processors in
  $a.members_g$) $p_j$'s time to live counter
$a.vid$: identifier for the current view of the
  group
Local variables of processor $p_i$:
$g_i$: boolean indicating whether or not $p_i$ is
  in the group
$A_i$: set of agents currently at processor $p_i$
$last.vid_g$: the value of the $vid$ for group $g$
  recorded at $p_i$

1. Upon Agent Arrival:
1.1     $\forall a.lv_j \in a.lvs\ a.lv_j \leftarrow a.lv_j - 1$
        $\{a$ is the arriving agent$\}$
1.2     $\mathcal{A}_i \leftarrow \mathcal{A}_i \cup a$
1.3     reset timeout clock

2. Upon Timeout $tp_i$:
2.1     $a \leftarrow$ create_an_agent
2.2     $\mathcal{A}_i \leftarrow a$
2.3     reset timeout clock

3. Execute an agent step:
3.1     if $|\mathcal{A}_i| > 1$ then
3.1.1       $\mathcal{A}_i \leftarrow$ create_an_agent
3.2     if $|\mathcal{A}_i| = 1$ then
3.2.1       remove the agent $a$ from $\mathcal{A}_i$
3.2.2       $members \leftarrow a.members_g$
3.2.3       $\forall a.lv_j \leq 0$
3.2.3.1         remove $j$ from $a.members_g$
3.2.3.2         remove $lv_j$ from $a.lvs$
3.2.4       if $g_i = true$ then
3.2.4.1         add $i$ to $a.members_g$
3.2.4.2         add $lv_i = ttl_i$ to $a.lvs$
3.2.5       else ($g_i = false$)
3.2.5.1         remove $i$ from $a.members_g$
3.2.6       if $members \neq a.members_g$ then
3.2.6.1         $a.vid \leftarrow a.vid + 1$ modulo $\mathcal{I}$
3.2.10      $last.vid_g \leftarrow a.vid$
3.2.11      choose $p_{next}$
3.2.12      send $a$ to $p_{next}$

f. Function create_an_agent:
f.1     $a.vid \leftarrow last.vid_g + 1$ modulo $\mathcal{I}$
f.1     if $g_i = true$ then
f.1.1       $a.members_g \leftarrow i$
f.1.2       $a.lv_i \leftarrow ttl_i$
f.2     else
f.2.1       $a.members_g \leftarrow \emptyset$
f.3     return $a$
```

Fig. 2. Self-stabilizing membership service, code for p_i.

one (line 1.1) and add a to $\mathcal{A}_i$ (line 1.2). Then, we reset the clock that measures tp_i time period (line 1.3). When tp_i expires, we create a new agent (line 2). We use the function *create_an_agent* for this task. A new agent is created when agents collide (line 3.1.1). The newly created agent replaces all the agents in $\mathcal{A}_i$. Line 3.2.2 stores the current list of members (later used in line 3.2.6). Lines 3.2.3 to 3.2.3.2 remove any node that has been flagged as disconnected. Lines 3.2.4 to 3.2.4.2 add i to the membership upon a request, while lines 3.2.5 to 3.2.5.1 remove i. Line 3.2.6 creates a new view if the list of members has changed. Lines 3.2.11 to 3.2.12 forward the agent to a randomly chosen neighbor. Lines f.1 to f.3 create a new agent.

Next, we prove that our algorithm satisfies the membership service requirements. Recall that we define a *nice execution* to be an execution in which 1) there exists a single agent and 2) the single agent visits every processor in the system within, at most, every M consecutive agent moves, where M is a constant (that depends on n).

Lemma 2. *Every nice execution of our algorithm satisfies Requirement 1 and Requirement 2.*

Proof. Let R be a nice execution, and a be the agent.

We first prove that Requirement 1 holds. Within M agent moves, a visits p_i.

Suppose that $g_i = true$ throughout. Then, in every visit of a to p_i, lines 3.2.4.1 and 3.2.4.2 are executed, and line 3.2.5.1 is not executed. Therefore, immediately after the first visit of the agent at p_i, it holds that p_i appears in $members_g$. The fact that $ttl_i \geq M$ implies that a visits p_i again before $a.lv_i \leq 0$. Therefore, after the first visit of a in p_i, lines 3.2.3.1 and 3.2.3.2 are not executed in R.

Suppose that $g_i = false$ throughout. Then, in every visit of a to p_i, lines 3.2.4.1 and 3.2.4.2 are not executed, and line 3.2.5.1 is executed.

Therefore, after the first visit of the agent to p_i, p_i does *not* appear in $members_g$.

The proof is completed since lines 3.2.3.1, 3.2.4.1, and 3.2.5.1 are the only lines in the algorithm that remove or add to $members$.

We now turn to prove that Requirement 2 holds as well. Suppose all g_i variables have fixed values. Since Requirement 1 holds, we can conclude that after M agent moves p_i appears in $members_g$ if, and only if, $g_i = true$.

Therefore, $a.members_g$ is fixed after M agent moves, and line 3.2.6.1 is not executed. Hence, both $a.vid$ and $a.members_g$ are fixed after M agent moves. $\square$

Note that p_i appears in (respectively,, is removed from) $a.members_g$ following ttl_i agent moves, in which the value of g_i is true (respectively, false). Thus, the time it takes to reach a legal execution in which the values in $a.members_g$ and $a.vid$ reflects a traversal of the agent in an *initialized execution*(an execution in which a single group exists and this group does not include any processor).

4.2 Accelerating Stabilization by Estimating n

We now show that it is possible to quickly estimate the actual number of nodes n in the connected component (see, e.g., [17] for a similar approach in the nondistributed case). This estimate can be used in place of the upper bound N when calculating the values of tp_i, lvs, and ttl_i. Having a more accurate upper bound on n will ensure that the system will react faster to changes such as addition/removal of a processor. Roughly speaking, the estimation algorithm is used to ensure the existence of a single agent faster, but at the same time requires more (short) messages to traverse the system simultaneously. There is a trade-off between the time it takes for estimating n and the frequency of sending scouters, which in turn influences the number of messages present in the system. Fast stabilization to a correct estimate requires frequent scouter creation, and conversely, less frequent scouter creation implies a longer stabilization period. We note that there are situations in which the number of scouters in the system can be dramatically reduced while the stabilization time is not changed much, i.e., if the frequency of scouter creation is reduced from every single time unit to every other time unit, then the number of scouters is approximately halved while the expected stabilization time is still of the same order.

As mentioned earlier, a processor p_i will send a special type of agent, called *scouter*, following a predefined (relatively short) time period in which no scouter visited p_i. A scouter can be viewed as a "light weight" agent that collects indications concerning the alive processors in the system. The briefcase of a scouter contains a set of identifiers of the processors that the scouter visits—this set is called $alive$. The briefcase of a scouter also includes, for each processor p_j in $alive$, a counter lv_j which counts the number of scouter moves since the last visit of the scouter to the processor p_j. The set of the lv counters of the processors in $alive$ is called lvs. Whenever a scouter moves from p_i to p_j, all the lvs counters are incremented by 1 except lv_j, which is set to zero.

The number of identifiers in the $alive$ set serves for estimating the number of processors in the system. The estimated number of processors in the system is distributed to the processors in the course of the scouters' random walks.

Roughly speaking, the lvs counters are kept small for existing processors and become large for nonexistent processors. The random walk ensures with high probability that existing processors will be visited often (every so many scouter steps, which is a function of the actual number of processors) and, therefore, will be assigned zero every so often. On the other hand, counters of nonexistent processors in lvs will be incremented up to $\ell(sc) \cdot CT(N+1)$. We define ℓ to be a safety function that uses the current scouter data to define the number of cover times required before removing processors from the $alive$ set of the scouter and before deciding that the scouter is mature (the maturity of a scouter is defined in the sequel). We define $CT(n)$ to be the expected cover time for a system with n processors.

The idea for estimating the number of active processors quickly is to sort the counters in lvs according to their values and then find a "too large" gap between two successive counter values; namely, a gap between the $(j_{k-1})\mathrm{th}$ and the $(j_k)\mathrm{th}$ smallest counters in the sorted lvs counters, such that $lv_{j_{k-1}} - lv_{j_k}$ is (much) larger than the expected cover time of a system with k processors.

The code of the estimation algorithm is presented in Fig. 3, and the code in Fig. 3 is similar to the code presented in Fig. 2; the changed code lines (1.1, 3.1.1 3.2.2 to 3.2.6, and f) are marked in bold. In line 1.1, the scouter increments by one the moves counter of every node with index in $alive$. Note that the counter value is bounded by the maximal number of moves that a scouter may take (without visiting p_j) before considering p_j to be not connected. This upper bound depends on the expected cover time, CT, and the safety function, ℓ. We note that the larger the value of ℓ, the more reliable the estimate for the number of active processors, but the longer it takes to stabilize.

Processor p_i resets the clock that measures the time period in between successive scouter visits in p_i (lines 1.3 and 2.3). The timeout clock reset command sets the clock by a predefined (constant) time period, which is defined by the system administrator.

In line 3.1.1, p_i chooses a single scouter among the scouters hosted by p_i. The function *First* returns a scouter $sc \in \mathcal{A}_i$, such that $sc.sm$ is maximal. If there are several such scouters, then p_i chooses one of them arbitrarily.

In lines 3.2.2 to 3.2.3, p_i adds its identifier to $alive$ and sets its corresponding moves counter to zero. In line 3.2.4, the *ConPrefix* function is executed. The function removes k from the $alive$ set using the following criteria: Let $lv_{j_1}, \cdots, lv_{j_m}$ be the increasingly ordered sequence of the counters in lvs. The scouter removes any j_m from $alive$, such that there exists a $k \leq m$, for which $lv_{j_k} - lv_{j_{k-1}}$ is greater than $\ell \cdot CT(k)$. This means we choose a gap in the lvs of the kth and $(k-1)\mathrm{th}$ elements, such that this gap is larger than the number of scouter moves required to explore (in a random walk fashion) a connected component of k processors. In line 3.2.5, the scouter moves counter, sm, is incremented by one. The moves counter value is bounded by the (expected) maximal cover time (in terms of number of moves) of a system with N processors. Newly created scouters visit only a portion of the system and, therefore, cannot have the estimated number of processors until they perform a large enough number of moves in which the set of processors visited is fixed. That is, a processor updates its estimate number of processors, only when the scouter has already taken $\ell(sc) \cdot CT(|alive|)$ moves (line 3.2.6). The size of $alive$ is assigned to the $estimated\_n$ variable of the visited processor.

```
Scouter Data Structure has fields:
$sc.alive$: set of processors in the system
$sc.lv_j$: (where the $p_j$ are the processors in
 $sc.alive$) $p_j$'s moves counter
$sc.sm$: counts scouter moves

Local variables of processor $p_i$:
$A_i$: set of scouters currently at $p_i$

1. Upon Scouter Arrival:
1.1  $\forall sc.lv_j \in sc.lvs\ sc.lv_j \leftarrow$
         $\min(sc.lv_j + 1, \ell(sc) \cdot CT(N+1))$
         $\{sc$ is the arriving scouter$\}$
1.2  $\mathcal{A}_i \leftarrow \mathcal{A}_i \cup sc$
1.3  reset timeout clock

2. Upon Timeout:
2.1  $sc \leftarrow$ create_a_scouter
2.2  $\mathcal{A}_i \leftarrow sc$
2.3  reset timeout clock

3. Execute a scouter step:
3.1  if $|\mathcal{A}_i| > 1$ then
3.1.1  $\mathcal{A}_i \leftarrow$ First$(\mathcal{A}_i)$
3.2  if $|\mathcal{A}_i| = 1$ then
3.2.1  remove the scouter $sc$ from $\mathcal{A}_i$
3.2.2  add i to sc.alive
3.2.3  add $lv_i$=0 to sc.lvs
3.2.4  sc $\leftarrow$ ConPrefix(sc)
3.2.5  sc.sm $\leftarrow \min(sc.sm + 1,$
           $\ell(N+1) \cdot CT(N+1))$
3.2.6  if $\ell(|sc.alive| + 1) \cdot CT(|sc.alive| + 1)$
           $\leq$ sc.sm then
3.2.6.1  sc.estimated_n $\leftarrow |sc.alive|$
3.2.7  choose $p_{next}$
3.2.8  send $sc$ to $p_{next}$

f. Function create_a_scouter:
f.1  sc.alive $\leftarrow \emptyset$
f.2  sc.mc $\leftarrow$ 0
f.3  return $sc$
```

Fig. 3. Algorithm for estimating n, code for p_i.

Lines f.1 to f.3 create a new scouter. In line f.1, p_i empties the $alive$ set. Line f.2 sets the scouter moves counter to zero. Line f.3 returns the newly created scouter.

We now turn to demonstrate the correctness of the algorithm presented in Fig. 3. We consider fair executions in which there exists a scouter sc in every configuration. Note that every fair execution has a suffix in which in every configuration there is a scouter; this suffix is reached (following a constant/short time period) at most after the first creation of a new scouter in the execution.

We consider a fair execution R in which the random walks of existing scouters behave as if the communication graph is fixed as discussed in Section 3 (other cases are analyzed analogously).

Lemma 3 shows that the value of $sc.lv_k$ is eventually the number of scouter moves that the scouter sc took since the last visit of sc to p_k.

Lemma 3. *Let R be a fair execution that starts in configuration c_0, and let c_1 be a later configuration of R. Let sc be a scouter in c_0 and in c_1. Suppose that $k \in sc.alive$ in c_0 and c_1. Let x be the value of $sc.lv_k$ in c_0, and y be the value of $sc.lv_k$ in c_1. Then, $y = \min(x + z, \ell(N+1) \cdot CT(N+1))$, where z is the number of moves sc takes between c_0 and c_1.*

Proof. By the algorithm presented in Fig. 3, only line 1.1 modifies $sc.lv_k$, increasing its value by one in every scouter move. Hence, the lemma. □

Lemma 4 is demonstrated by arguments similar to Lemma 3, this time considering the moves counter of a scouter.

Lemma 4. *Let R be a fair execution that starts in configuration c_0 and let c_1 be a later configuration of R. Let sc be a scouter in c_0 and c_1. Let x be the value of $sc.mc$ in c_0 and y be the value of $sc.mc$ in c_1. Then, $y = \min(x + z, \ell(N+1) \cdot CT(N+1))$, where z is the number of moves sc takes between c_0 and c_1.*

Lemma 4 implies that a newly created scouter sc cannot distribute an estimate (i.e., execute line 3.2.6.1), before taking $\ell(|sc.alive| + 1) \cdot CT(|sc.alive| + 1)$ scouter moves.

In the next two lemmas, we assume the existence of a single scouter. Later, we extend the results to the case in which there are at most $n \leq N$ scouters in the system.

Lemma 5 shows that the scouter does not remove identifiers of processors that it does visit.

Lemma 5. *Let R be a fair execution that starts in configuration c, and let sc be a scouter in R. For every (positive integers) n and $x > 1$, there exists ℓ such that in a suffix R' of R, that immediately follows the first $\ell \cdot CT(n)$ scouter moves of sc in R, the probability for every $p_j \in \mathcal{V}$ that j will not be in $sc.alive$ is less than 2^{-x}.*

Proof. We compute the probability of the existence of a configuration c' in R' in which j does not exist in $sc.alive$. The expected cover time of the system is $CT(n)$; thus, the probability that a processor in the system is not visited in the $\ell \cdot CT(n)$ scouter moves that immediately precede c' is $2^{-\ell}$.

In case p_j is visited during these moves, p_j may be removed later due to the execution of $ConPrefix$. Consider the configuration c_1 in which the scouter visits p_j for the last time prior to c'. p_j may be removed from sc only if there is a (large) gap in the lv values of the processors that are visited in the execution that starts in

c_1 and ends in c'. Let m be the number of distinct processors visited following c_1 and before c'. We now analyze the probability for such a gap to occur.

We now choose ℓ to be a function of k as follows: $\ell(k) = 2\log k$.

Let $P_r(k, \ell(k))$, $1 \leq k \leq m$, be the probability that sc takes more than $\ell(k) \cdot CT(k)$ scouter moves, while visiting a set of $k-1$ processors and not visiting any (of the $n-k+1 \geq m$) other processors. This is in fact the probability that *ConPrefix* will remove existing processors due to a gap between the $k-1$th and the kth lv in lvs. We will now accumulate the probabilities of such events for each $1 \leq k \leq m$.

Note that sc adds its current host to $alive$ (in lines 3.2.2 and 3.2.3), thus it is obvious that $P_r(1, \ell(1)) = 0$. Also, recall that the expected cover time of k processors is $CT(k)$; hence, the probability of covering k processors in $CT(k)$ scouter moves implies that $P_r(k,1) < 2^{-1}$, and $P_r(k, \ell(k)) < 2^{-\ell(k)}$. Since we choose $\ell(k) = 2\log k$, we have $P_r(k, 2\log k) \leq 1/2^{\log k^2} = 1/k^2$, for any $1 < k \leq m$. We now compute the probability that for some $1 \leq k \leq m$, *ConPrefix* will remove existing processors from $alive$. Since we choose $\ell(k) = 2\log k$, we have $\Sigma_{k=1}^{m} P_r(k, 2\log k) = \Sigma_{k=2}^{m} 1/k^2 < 1/2$. To have a probability smaller than 2^{-y} for the removal of p_j, we may choose $\ell(k) = 2y\log k$, which implies

$$\Sigma_{k=1}^{m} P_r(k, 2y\log k) = \Sigma_{k=2}^{m} (1/k^2)^y < 1/2^y.$$

The value of x is a function of y that can be determined in a straight-forward manner. □

Lemma 6 completes the proof of the algorithm presented in Fig. 3 by showing that identifiers of nonexistent processors are quickly removed from the $alive$ set of scouters.

Lemma 6. *Let R be a fair execution that starts in configuration c, and let sc be a scouter in R. For every (positive integers) n and $x > 1$ there exists ℓ, such that, in a suffix R' of R that immediately follows the first $2\ell \cdot CT(n)$ scouter moves of sc in R, the probability for every $j \in sc.alive$ that p_j will not be in $\mathcal{V}$ is less than 2^{-x}.*

Proof. For every configuration c' in R', the set $alive(c')$ is the set of identifiers that are in both the sets $sc.alive$ and $\mathcal{V}$ of configuration c'. Define $max\_alive(c')$ to be $\max_{m \in alive(c')} sc.lv_m$.

Let $floating(c')$ be the set of identifiers of processors that are not in $\mathcal{V}$ that appear in the $sc.alive$ set of configuration c'. Define $min\_floating(c')$ to be

$$\min_{m \in floating(c')} sc.lv_m.$$

We now analyze the probability for $floating(c')$ to be nonempty.

Note that the set $floating(c)$ of the first configuration c of R is a superset of any $floating(c'')$ where c'' is a configuration in R. sc does not visit any member of $floating(c')$ during R; thus, in every scouter move from c to c', the lv of every $floating(c')$ is incremented by 1.

By Lemma 3, the fact that $floating(c') \neq \emptyset$ implies that $min\_floating(c')$ equals

$$\min(min\_floating(c) + z, \ell(N+1) \cdot CT(N+1)),$$

where z is the number of moves sc takes between c and c'. In particular, $min\_floating(c')$ is greater than $2\ell \cdot CT(n)$. By Lemma 5, the probability that $max\_alive(c')$ is smaller than $\ell \cdot CT(n)$ is no more than $1/2^x$. Therefore, the probability that line 3.2.4 does not remove all the members of $floating(c')$ from $sc.alive$ is less than $1/2^x$. □

Corollary. *For every $x > 1$, there exists a logarithmic function ℓ of the maximum number of scouters (one scouter for each existing processor) in the system, such that the probability that every scouter sc notifies each processor with the number of processors in the system within $O(l \cdot CT)$ moves of sc is $1 - 2^{-x}$.*

Proof. Lemma 3 and Lemma 6 assumed a single scouter. In case we have k scouters, the probability that $sc.alive$ is not identical to $\mathcal{V}$ is increased. We have to choose the ℓ function in both lemmas such that the probability for failure of a single scouter is 2^{-y}. Thus, the probability that at least one of the k scouters fails is no greater than $k2^{-y}$, and when $y = x\log k$, then $k2^{-y} = k2^{-x\log k} = k2^{-x}/k = 2^{-x}$, which completes the proof. □

5 Group Multicast

In this section, we show how the membership service described in Section 4 can be used to support multicast services.

Past work on total ordering has yielded several approaches that use a *token* that traverses a (virtual) ring, to implement the total order. A popular approach is to continually circulate a token through all the nodes of the network in a *virtual ring* (e.g., [31], [3]). The token circulates around the virtual ring carrying a sequence number. When a node receives the token, it assigns sequence numbers (carried with the token) to its messages, and then multicasts the messages to the group members. The sequence number carried in the token is incremented once for each message sent by the node holding the token. Since the messages are assigned globally unique sequence numbers, total order can be achieved. (Additional mechanisms are needed, depending on the desired level of reliability.) An alternative approach (e.g., [18], [10]) is to store the messages in the token itself—since the token visits all nodes in a virtual ring, the messages will eventually reach all the nodes, the order in which messages are added to the token determining the order in which they are delivered to the nodes.

Here, we use a scheme in which the agent carries the messages. Any processor p_i that wishes to multicast a message m, waits for the membership agent and augments it with the multicast message.

5.1 Group Multicast Requirements

Let R be a nice execution of the membership algorithm presented in Fig. 2.

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*

```
3.2.6.2     $a.history \leftarrow a.history + (a.vid, members)$
3.2.7       if $p_i$ wishes to send a multicast message $m$ then
3.2.7.1         add $m$ to $a.messages$
3.2.8       for every non delivered $v \in a.history$
                application_layer $\leftarrow v$
3.2.9       for every non delivered $m \in a.messages$
                application_layer $\leftarrow m$
```

Fig. 4. Self-stabilizing multicast service by random walk.

p_i before m_1 is delivered, then m_0 is delivered to p_j before m_1 is delivered.

Since we have a single agent, we can accumulate the history of the membership views and the multicast messages within each view in the agent. The views and messages are stored in the order that they were sent, and delivered in a first-in first-out manner. Note that the group membership service notifies the members of the group with the view history as part of the service. The view history is important for the application, for example, to assist the application to know whether there is a view of which it is not a member. In this case, the messages sent in the immediately preceding view in which it was a member may not have reached it.

Whenever an agent arrives at a processor, the processor can receive all the multicast messages that are related to views of which it is a member. Moreover, the processor can deliver these messages in order and with the appropriate view identifier.

A view of a group becomes *old* when a new view is established (by one of the processors) for the same group. An old view $view_o$ and the multicast messages (within this view) are removed from the agent a, when, for every processor p_i that is a member of $view_o$, the multicast messages of this view have been delivered to p_i, or when there is an indication that p_i is not in the same connected component as the agent a.

We call the above multicast service *best effort* multicast. We note that the multicast service is optimal if old views are not eliminated from the agent before all the members receive the multicast messages, except the ones that are not present in the system (and we do not know whether they will be reconnected).

The history length is bounded; the bound is a function of the maximal activity in the system in terms of multicast and view establishments during kN^3 agent moves. Note that an old view is eliminated when there are kN^3 agent moves following the establishment of a new view. The reason is that either every processor in the old view is visited or is considered not connected. In addition, the current view may accumulate, at most, kN^3 multicast messages (a message in every agent move) when all the processors in the view are considered connected and active.

The formal description of the multicast algorithm appears in Fig. 4. This description extends the code of Fig. 2. A new view is added to the $history$ of an agent a, upon the $history$ creation (line 3.2.6.2). If processor p_i wishes to send a multicast message m, then m is added to the $messages$ of a (line 3.2.7.1). Every view (respectively, message) that processor p_i has not yet received is delivered to the application layer in line 3.2.8 (respectively, 3.2.9).

It is possible to extend the multicast service to support indication of the delivery to all the processors in the group (in the spirit of *safe delivery* [7]) and an indication of the fact that all the processors are aware of the current view (in the spirit of *view agreement* [7]). The idea is to add an indication for each delivery of a message or a view to a processor, and use these indications to conclude safe delivery or view agreement.

Next, we prove that our multicast service algorithm satisfies Requirements 3 and 4.

Lemma 7. *Every nice execution of our algorithm satisfies Requirement 3 and Requirement 4.*

Proof. We first prove that Requirement 3 holds. A message m sent by p_j is not removed from the agent for kN^3 agent moves. Clearly, if a visits p_i during these kN^3 agent moves, then m is delivered to p_i. Since p_i must choose $ttl_i \leq kN^3$, and p_i is not removed from v during R, then the agent must arrive at p_i following the delivery of m and before it is removed.

We now show that Requirement 4 holds. Send operations are executed during the visit of the (single) agent and therefore can be (totally) ordered. Assume that m_0 is sent in R before m_1 is sent and let p_i and p_j be two processors that deliver m_0. Every processor p_i that receives an agent and delivers m_0, either finds m_1 in the agent as well (in this case, m_1 has been sent before m_0 is delivered by p_i) and delivers m_0 and then m_1, or p_i does not find m_1 in the agent (in this case, m_1 has not yet been sent) and delivers m_1 only in a subsequent visit of the agent—a visit that follows the delivery of m_0. Thus, the order of delivery of the messages m_0 and m_1 by every processor p is identical to the order of the send operations of m_0 and m_1. □

Up to this point, we always assumed that there exists a single agent in the system and used an "empty" agent to replace colliding agents. Let us remark that a technique similar to the one presented in [14] can be used to resolve history conflicts upon agent collisions and decide on a single nonempty history.

6 Resource Allocation

The random walk of the agent and the membership service can support not only a multicast service, but also another application—a resource allocation service. For the sake of simplicity, we assume that there is no interaction between different resources. In other words, we handle a single resource in the system.

The problem of resource allocation has been extensively studied (e.g., [25], [8], [13]). In [32], the task of resource allocation is considered in the context of group communication: three different group membership protocols are used to solve a resource allocation problem named Bancomat. The different solutions vary in communication characterizations and their ability to decide independently. The design of [32] is for a complicated resource allocation task, but is not self-stabilizing. Here, in contrast, we present a self-stabilizing solution for a basic resource allocation task.

6.1 Resource Allocation Requirement

Requirement 5. *Let R be a nice execution, such that every processor that possesses the resource releases it after B asynchronous cycles, for some finite constant B. Then, 1) every processor p_i that wishes to possess the resource infinitely often possesses the resource infinitely often and 2) in any configuration at most one processor possesses the resource.*

The communication graph of an ad hoc system may be partitioned into multiple mutually disconnected connected components. Here, we describe an algorithm for resource allocation, despite such dynamic communication graph partition.

Group membership services have two approaches for coping with partition scenarios. *Partitionable* membership services allow multiple disjoint views of the same group to exist concurrently, each view for a different component [7].

In contrast, *primary component* membership services allow only one component, called the primary component, to have group views and the full set of allowed operations, while other components are considered to be nonprimary and are limited to executing a reduced set of operations [7].

We note that the self-stabilization property imposes the requirement that the number of processors of any primary view must include the majority of processors in the system ($\lfloor N/2 \rfloor + 1$). Suppose that we do not require primary views to include the majority of processors and that, for every set of processors, there is an execution in which this set of processors (perhaps the only active processors) forms a primary view. Consider an execution R_1 in which A is a primary view that consists of a set of processors, and consider a different execution R_2 in which A' is a primary view that consists of a totally disjoint set of processors. Note that, by our assumption, any set of processors can form a primary view. Consider an execution R_3 with two disjoint "primary" connected components A and A' in the first configuration of an execution R_3. Since there is no communication between the two components, the system may never detect the fact that both components are considered primary.

We define a group, g_{all}, that includes all the processors. This group will be used to indicate whether the connected component is a primary component. The agent has a Boolean flag $a.primary$ that is true if, and only if, the number of members in g_{all} is greater than $\lfloor N/2 \rfloor$. The agent decides on the list of processors in the connected component by the membership procedure described above.

Processors that request the resource join the group $g_{resource}$. The agent can order the processors in the $g_{resource}$ members set by the order in which they join the set; in this case, the set is essentially a request queue.

The agent a of the primary component allocates the resource to the processor p_r that is at the head of the request queue. The resource is released when p_r leaves $g_{resource}$, p_r leaves g_{all}, or $a.primary$ is false.

The next lemma proves that our algorithm satisfies the resource allocation requirement.

Lemma 8. *Every nice execution of our algorithm satisfies Requirement 5.*

Proof. Let p_i be a processor that wishes to possess the resource in R. Then, by the algorithm it joins $g_{resource}$. Since 1) $g_{resource}$ is a queue with, at most, N requests, 2) the agent cover time is bounded by M, and 3) the time that a processor possesses the resource is bounded by B, then within $O(MNB)$ asynchronous cycles, the agent allocates the resource to p_i. □

7 Concluding Remarks

We suggest using a random walk of an agent to cope with the uncertainty and the dynamic nature of ad hoc networks. The random walk of the agent is used to implement a *probabilistic group communication service*. The membership service, multicast service, and resource allocation service that we present meet their requirements with high probability. We emphasize that the communication, time and space resources for operations can be tuned by varying the probability. The requirements will hold with higher probability if we increase the value of the transmission radius rd, enlarge the parameter k for ensuring cover time, and use longer histories in the agents.

We argue that our new approach for a best effort service matches the nature of the ad hoc system and the limitations (e.g., [5], [28]) of the group communication service. The traversal of the system by a single agent limits the number of simultaneous messages that are needed to support the group communication service at any given time. Thus, it limits the resources (processing capabilities) of the mobile agent needed to support these services.

Acknowledgments

The authors would like to thank Uriel Feige for helpful discussions, Lyn Pierce for improving the presentation, and Nir Alfasi for performing the simulations. This work was partially supported by US National Science Foundation Award CCR-0098305, Texas Advanced Research Program Award 00512-0091-2001, and Rita Altura Trust Chair. An extended abstract of this work was presented at the 21st IEEE Symposium on Reliable Distributed Systems, Japan, 2002.

References

[1] D. Aldous and J.A. Fill, *Reversible Markov Chains and Random Walks on Graphs,* Oct. 1999, http://www.stat.berkeley.edu/~aldous/book.html.

[2] N. Alfasi, "Cover Time Random Walk in Ad Hoc Networks," Technical Report #03-02, Dept. of Computer Science, Ben-Gurion Univ. of the Negev, 2002.

[3] Y. Amir, L. Moser, P.M. Melliar-Smith, D. Agrawal, and P. Ciarfella, "Fast Message Ordering and Membership Using a Logical Token-Passing Ring," *Proc. 13th IEEE Int'l Conf. Distributed Computing Systems,* pp. 551-560, 1993.

[4] J. Beauquier, T. Herault, and E. Schiller, "Easy Self-Stabilization with an Agent" *Proc. Fifth Workshop Self-Stabilizing Systems,* pp. 35-50, 2001.

[5] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the Impossibility of Group Membership," *Proc. ACM Symp. Principles of Distributed Computing,* pp. 322-330, 1996.

[6] I. Chatzigiannakis, S. Nikoletseas, and P. Spirakis, "An Efficient Communication Strategy for Ad hoc Mobile Networks," *Proc. 15th Int'l Symp. Distributed Computing,* pp. 285-299, 2001.

[7] G.V. Chockler, I. Keidar, and R. Vitenberg, "Group Communication Specifications: A Comprehensive Study," *ACM Computing Surveys,* vol. 33, no. 4, pp. 1-43, Dec. 2001.

[8] M. Choy and A.K. Singh, "Efficient Fault Tolerant Algorithms for Distributed Resource Allocation," *ACM Trans. Programming Languages and Systems,* vol. 17, no. 4, pp. 535-559, 1995.

[9] D. Coppersmith, P. Tetali, and P. Winkler, "Collisions among Random Walks on a Graph," *SIAM J. Discrete Math,* vol. 6, no. 3, pp. 363-374, Aug. 1993.

[10] F. Cristian and F. Schmuck, "Agreeing on Processor Group Membership in Asynchronous Distributed Systems," Technical Report CSE95-428, Dept. of Computer Science, Univ. of California, San Diego, 1995.

[11] R.A. Dayem, *Mobile Data and Wireless LAN Technologies.* Prentice Hall, 1997.

[12] E.W. Dijkstra, "Self Stabilizing Systems in Spite of Distributed Control," *Comm. ACM,* vol. 17, pp. 643-644, 1974.

[13] S. Dolev, *Self-Stabilization.* MIT Press, 2000.

[14] S. Dolev and E. Schiller, "Communication Adaptive Self-Stabilizing Group Membership Service," *IEEE Trans. Parallel Distributed Systems,* vol. 14, no. 7, pp. 709-720, 2003.

[15] U. Feige, "A Tight Upper Bound on the Cover Time for Random Walks on Graphs," *Random Structures and Algorithms,* vol. 6, no. 4, pp. 51-54, 1995.

[16] U. Feige, "A Tight Lower Bound on the Cover Time for Random Walks on Graphs," *Random Structures and Algorithms,* vol. 6, no. 4, pp. 433-438, 1995.

[17] U. Feige, "A Fast Randomized LOGSPACE Algorithm for Graph Connectivity," *Theoretical Computer Science,* vol. 169, pp. 147-160, 1996.

[18] A. Fekete, N. Lynch, and A. Shvartsman, "Specifying and Using a Partitionable Group Communication Service," *ACM Trans. Computer Systems,* vol. 19, no. 2, pp. 171-216, 2001.

[19] S. Ghosh, "Agents, Distributed Algorithms, and Stabilization," *Computing and Combinatorics,* pp. 242-251, 2000.

[20] K.P. Hatzis, G.P. Pentaris, P.G. Spirakis, V.T. Tampakas, and R.B. Tan, "Fundamental Control Algorithms in Mobile Networks," *Proc. 11th ACM Symp. Parallel Algorithms and Architectures,* pp. 251-260, 1999.

[21] T. Herman and T. Masuzawa, "Self-Stabilizing Agent Traversal," *Proc. Fifth Workshop Self-Stabilizing Systems,* pp. 152-166, 2001.

[22] IETF Mobile Ad hoc Networks (MANET) Working Group, http://www.ietf.org/html.charters/manet-charter.html, 2005.

[23] A. Israeli and M. Jalfon, "Token Management Schemes and Random Walks Yield Self-Stabilizing Mutual Exclusion," *Proc. Ninth Ann. ACM Symp. Principles of Distributed Computing,* pp. 119-131, 1990.

[24] T. Imielinski and H.F. Korth, *Mobile Computing.* Academic Publishers, 1996.

[25] N.A. Lynch, "Fast Allocation of Nearby Resources in a Distributed System," *Proc. 12th ACM Symp. Theory of Computing,* pp. 70-81, 1980.

[26] L. Lovasz, "Random Walks on Graphs: A Survey," *Combinatorics, Paul Erdos is Eighty,* vol. 2, pp. 353-398, Budapest: Janos Bolyai Mathematical Society, 1996.

[27] M. Mihail and C.H. Papadimitriou, "On the Random Walk Method for Protocol Testing," *Proc. Conf. Computer Aided Verification,* pp. 132-141, 1994.

[28] G. Neiger, "A New Look at Membership Service," *Proc. 15th ACM Symp. Principles of Distributed Computing,* pp. 331-340, 1996.

[29] NS2 Network Simulator, http://www.isi.edu/nsnam/ns/, 1989.

[30] V.D. Park and M.S. Corson, "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks," *Proc. IEEE INFOCOM Conf.,* pp. 1405-1413, Apr. 1997.

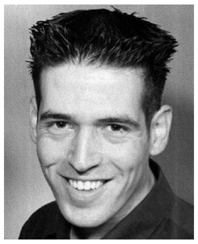
[31] B. Rajagopalan and P. McKinley, "A Token-Based Protocol for Reliable, Ordered Multicast Communication," *Proc. Eighth IEEE Symp. Reliable Distributed Systems,* pp. 84-93, Oct. 1989.

[32] J. Sussman and K. Marzullo, "The Bancomat Problem: An Example of Resource Allocation in a Partitionable Asynchronous System," *Proc. 12th Int'l Symp. Distributed Computing (DISC),* 1998.

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

[34] B. Lange and Y. Aridor, "Agent Transfer Protocol—ATP/0.1," 1999, http://www.trl.ibm.com/aglets/atp/atp.htm.

Shlomi Dolev received the BSc degree in engineering and the BA degree in computer science in 1984 and 1985, and the MSc and DSc degrees in computer science in 1990 and 1992 from the Technion Israel Institute of Technology. From 1992 to 1995, he was at Texas A&M University as a research specialist. In 1995, he joined the Department of Mathematics and Computer Science at Ben-Gurion University. He is the author of the book entitled *Self-Stabilization* (MIT Press, 2000). He is the founder and the first department head of the Computer Science Department at Ben-Gurion University. His publications, 100 conference and journal papers, are mostly in the area of distributed computing, communication networks, and security and cryptography; in particular, the self-stabilization property of such systems. Several agencies and companies support his research, including IBM (faculty awards), Intel, NSF, Deutsche Telekom, and the Israeli Ministries of Science and Defense. During his stay at Ben-Gurion University, he had visiting positions at several institutions including LRI, DIMACS, and MIT. He served on 30 program committees including a few of PODC, DISC, WSS, ICDCS, INFOCOM, and WADS. He is an associate editor of the *AIAA Journal of Aerospace Computing, Information, and Communication*. He holds the Ben-Gurion University Rita Altura trust chair in computer sciences. He is a senior member of the IEEE and the IEEE Computer Society.



Elad Schiller received the BSc (1998) and MSc (2000) degrees in mathematics and computer science from the Department of Mathematics and Computer Science at Ben-Gurion University of the Negev. Recently, he received the PhD degree (2004) from the Department of Computer Science at Ben-Gurion University of the Negev. After his PhD, he visited MIT and CTI (Greece).

Jennifer L. Welch received the BA from the University of Texas at Austin and the SM and PhD degrees from the Massachusetts Institute of Technology. She is currently holder of the Chevron II Professorship in the Department of Computer Science at Texas A&M University. She has also been on the faculty at the University of North Carolina and was a member of Technical Staff at GTE Laboratories Incorporated. Her research interests are in the theory of distributed computing, algorithm analysis, distributed systems, mobile ad hoc networks, and distributed data structures.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.