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

(Extended Abstract)

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

Jennifer Welch
Department of Computer Science,
Texas A&M University, USA.
welch@cs.tamu.edu

Abstract

We introduce a self-stabilizing group communication system for ad-hoc networks. The system design is based on random walks of mobile agents. 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. The new techniques support group membership and multicast, and also support resource allocation.

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 [10]. *Ad-hoc networks* [22, 23] do not use pre-existing 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 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, 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.

Fault tolerance is very important in distributed systems that may experience crashes of processors, 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 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 solves 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 will focus on algorithms that fit the special characteristics of ad-hoc networks. Mobile communication networks, by definition, experience movement of

*Partially supported by NSF Award CCR-0098305.

1060-9857/02 $17.00 © 2002 IEEE

some (or all) of the computing entities. Group communication services, in such a dynamic environment, 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 with 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 processor. 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 *agent* is responsible for broadcasting.

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 [26]). 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 [5] that assist in transferring information using random walks. In [5] 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). 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 [19] and then in e.g., [3, 21]. To the best of our knowledge, previous works considered fixed communication graphs, and did not address 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. 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 an 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 on the mobility pattern exhibited by an ad-hoc network. The group membership algorithm is presented in Section 4. The group multicast algorithm is presented 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, respectively, operation of a step. An *agent* is a program coupled with a program state (the program state is also called *briefcase*). In contrast, a token does not carry a program to be executed. The use of agents allows us to change the algorithm on the fly without re-programming 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 overwritten on the agent program.

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.

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}. 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 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 r 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 $neighbors_i$ set, such that the geographical distance between p_i and $p_j \in neighbors_i$ is no more than some r. $neighbors_i$ is symmetric, that is $p_j \in neighbors_i$ implies $p_i \in neighbors_j$. 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* $E = c_0, s_0, c_1, s_1, \ldots$ as an alternating sequence of system configurations c_i and steps s_i, such that each configuration c_{i+1} (except the initial configuration c_0) is obtained from its former configuration c_i by the execution of the step s_i. Note that s_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, s_i may cause a change in the communication graph. Thus, the only components that can be changed due to the execution of s_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 an execution in which: (1) there exists a single agent, and (2) the single agent arrives at every processor in at most every M consecutive moves, where M is a constant.

In the sequel (in Section 3) we compute the probability of having a nice execution in several common cases. The probability is calculated assuming an arbitrary configuration 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 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 legal executions *LE*. 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 require that a self-stabilizing ad-hoc system will satisfy: (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.

An ad-hoc self-stabilizing system recovers from transient faults that disturb its behavior for a limited period of time. The correctness of self-stabilizing ad-hoc systems is demonstrated by considering every nice execution that starts in a configuration 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 E is the shortest prefix, E', in which every processor executes at least one step. The second *asynchronous cycle* of E is the first *asynchronous cycle* of E'' where $E = E'E''$. The next asynchronous cycles are defined analogously.

3 Random Walks of Agents

The dynamic nature of ad-hoc networks does not allow us 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]) 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}, among the processors that it can directly communicate with. Then p sends the agent to p_{next}.

The above simple random walk procedure is used for covering the graph (broadcasting). We define the *cover time* of a graph to be the expected number of movements by a single agent that is required in order to visit every processor at least once. For calculating the cover time we choose a starting point that results in the maximal value for the cover time*. 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 cover time can be described in terms of asynchronous cycles instead of agent movements.

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 pre-defined 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 *meeting time* is an upper bound on the expected number of asynchronous cycles until a single agent exists. Note that the 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 meeting and 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). Then we present common cases in which the random walk succeeds in visiting all the processors. In the later 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

Suppose that processors p_1, p_2 and p_3 are connected in a ring topology. Assume that p_2 frequently moves towards p_1, losing connection with p_3, and then moves back to p_3, reestablishing the connection with p_3 and losing 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 detail, 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.

3.2 Viable communication graph

Here we consider the case where an agent infinitely often covers the system.

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

We say that the link between processors p_i and p_j is *viable* in an execution E if and only if (1) an agent visits both p_i and p_j infinitely often and (2) $p_j \in neighbors_i$ in an infinite number of visits of the agent in p_i and, (3) $p_i \in neighbors_j$ in an infinite number of visits of the agent in p_j.

We define $\mathcal{T} = p_{i_1}, \cdots, p_{i_l}$ to be a *viable path* between p_{i_0} and p_{i_l} if the links between p_{i_j} and $p_{i_{j+1}}$ are viable (where $p_{i_j} \in \mathcal{T}$ and $1 \leq j \leq l-1$).

We note that unless there is a viable path between p_i and p_j, 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 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 r can influence the frequency of changes in the communication graph $G(\mathcal{V}, \mathcal{E})$. At one extreme, r 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 then only very few changes in the communication graph are allowed. Another consideration is the velocity of processors with relation to the velocity 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 in 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 traversal does not visit a node more than once. 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 the meeting time and cover time of fixed graphs. The meeting time and the cover

time of a fixed graph are well studied. In [8] it is shown that within $O(n^3)$ agent movements, there is a single agent a. In [15] it is shown that the cover time is $O(n^3)$ agent moves.

3.4 Random changes in the graph

Here we assume the other extreme where the graph is always connected but can be totally changed in between two successive moves of the agent. This 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 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 movements, there is a single agent a. Thus, the expected meeting time of n agents is $O(n \log n)$ agent moves.

Lemma 3.1 *Let E be a fair execution, such that there are $k > 1$ agents in the first configuration of E, and no other agent is produced in E. 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 a single agent.

Suppose that a is the first agent to move in E. Then the probability that a chooses to move to processor p_i that hosts another agent is $(k-1)/(n-1)$. Therefore, $(n-1)/(k-1)$ is the expected number of agent movements in E before we have a configuration with $k-1$ agents. Since for every $1 < k \leq n$ we have $(k-1)/(n-1) > 0$, then the expected number of agent movements it takes for k agents to collide to 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 (maximal) 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. 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 on-line. p_i may count during a particular time period the number of times each neighbor has been connected to it and use these numbers to estimate q_j.

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 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 in g ($|members_g| \leq N$) 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 big enough number V that ensures the ordering between the existing views in the system. Views are eliminated from the system following a certain time interval (more details follow).

Group Membership Requirements:

Requirement 4.1 *For every nice execution, E, and every $p_i \in \mathcal{P}$ if g_i has a fixed value during E, then E has a suffix such that p_i appears in $members_g$ if and only if $g_i = true$.*

Requirement 4.2 *Every nice execution in which all the g_i variables have fixed values has a suffix such that 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 approach (see e.g., [30]). The agent carries a list, $members_g$, of members in the group g and a list of corresponding counters lvs — a counter value, lv_i, for each $p_i \in members_g$. 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 for covering the communication graph; this value can be changed by p_i when the estimate of the 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 $O(N^3)$. Cases in which the expected cover time is smaller or bigger are handled analogously. We suggest using 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, but 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 a single 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 suggest using 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.

A processor, p_i, uses a long time-out period tp_i (that is a function of the cover time for a graph of N nodes) to produce an agent.

A processor p_i for which $\mathcal{A}_i = \emptyset$ measures the period of time since the last time an agent visited p_i. p_i produces a new agent when the above period of time is greater than tp_i [†]. We note that the case of multiple agent creation is easily handled using agent collisions as we next describe. The value of $members_g$ of the new agent includes only i, the identifier of the processor that created the agent, and the value of lv_i.

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. The collision occurs when the agents are listed together in the set $\mathcal{A}_i$ of a processor p_i. For simplicity we assume that the value of $members_g$ of the new agent includes only the identifier of the processor, p_i, that created the agent and the value of lv_i.

Whenever 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 Figure 1. Upon agent a's arrival, we decrement every lv counter by one (line 1.1) and add a to $\mathcal{A}_i$ (line 1.2). 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 removes a node that has been flagged as disconnected. Lines 3.2.4 to 3.2.4.2 (3.2.5 to 3.2.5.1) adds (removes, respectively) i to (from) members upon a request. 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.2.3 create a new agent.

Next we prove that our algorithm satisfies the member-

[†]To avoid simultaneous creation of many agents one may choose tp_i to be also a function of the identifier of p_i, for example in case the expected cover time is C and the identifiers are small (say in the range 1 to N) then we may choose to assign $tp_i := i \cdot C$, where i is the identifier of p_i.

Global Constants:
ttl_i: time to live, k times the expected cover time
V: upper bound on the number of view identifiers that can be concurrently active
Agent Data Structure has Fields:
$a.members_g$: the set of processors in group g
$a.lv_j$: (where p_j are the processors in $a.members_g$) counter for the time to live of p_j
$a.vid$: 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 for-every $a.lv_i \in a.lvs$ $a.lv_i \leftarrow a.lv_i - 1$ {a is the arriving agent}
1.2 $\mathcal{A}_i \leftarrow \mathcal{A}_i \cup a$
1.3 reset timeout

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

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 for every $a.lv_j \leq 0$
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 V
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 if $g_i = true$ then
f.2.1 $a.vid \leftarrow last.vid_g + 1$ modulo V
f.2.1 $a.members_g \leftarrow i$
f.2.2 $a.lv_i \leftarrow ttl_i$
f.2.3 return a

Figure 1. Self-stabilizing Membership Service by Random Walk, Code for p_i

ship 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 processors in the system within at most every M consecutive moves, where M is a constant.

Lemma 4.3 *Every nice execution of our algorithm satisfies requirement 4.1 and requirement 4.2.*

Proof: Let E be a nice execution, and a be the agent.

We first prove that requirement 4.1 holds. Within M agent steps, 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 E.

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 4.2 holds as well. Suppose all g_i variables have fixed values. Since requirement 4.1 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. ■

Note that p_i appears in (resp., is removed from) $a.members_g$ following ttl_i agent moves, in which the value of g_i is true (resp., 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).

Accelerating stabilization by estimating n:

We now show that is possible to estimate the actual number of nodes n in the connected component (see e.g., [17]) and not use 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.

The briefcase of the agent includes a list of (no more than N) indices and time stamps (step counters), such that each index is associated with a time stamp. The list is sorted such that the most recently visited processor appears first. In other words, the index of node p_i is the tth element in the list, if the agent visited $t-1$ distinct processors following its last visit to p_i. To estimate the current number of processors, n, we suggest ignoring the suffix of the list that starts at the t'th element in the list such that there is a large enough gap between the time stamps of the t'th and the $t-1$'th elements in the list. Roughly speaking, we choose a gap in the time stamps of the t'th and $(t-1)'th$ elements such that this gap is larger than the expected number of steps required to explore (in a random walk fashion) a connected component of t processors.

5 Group Multicast

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

Past work on total ordering has yielded several approaches which use a *token* that traverses a (virtual) ring, to implement the total order. These algorithms have two approaches, one in which totally ordered message delivery is achieved by continually circulating a *token* through all the nodes of the network in a *virtual ring* (e.g., [27, 2]). 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, 9]) 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.

Group Multicast Requirements: Let E be a nice execution of the membership algorithm presented in Figure 1.

Requirement 5.1 *Suppose that processor p_i is a member of every view in E. If m is a message sent by a member of g during E, then m is delivered to p_i.*

Requirement 5.2 *Suppose that the messages m_0 and m_1 are delivered to processors p_i and p_j during E. If m_0 is delivered to 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.

Whenever an agent arrives at a processor the processor can receive all the multicast messages that are related to

```
3.2.6.2     a.history ← 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 ∈ a.history
                application_layer ← v
3.2.9       for every non delivered m ∈ a.messages
                application_layer ← m
```

Figure 2. Self-stabilizing Multicast Service by Random Walk

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 has been established to 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 in $view_o$ the multicast messages of this view have been delivered to p_i, or there is an indication that p_i is not in the same connected component with 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 unless all the members received the multicast messages (ignoring indication of non-connectivity).

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 steps 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 Figure 2. This description extends the code of Figure 1. 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 (resp., message) that processor p_i has not yet received, is delivered to the application layer in line 3.2.8 (resp., 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* [6]) and an indication of the fact that all the processors are aware of the current view (in the spirit of *view agreement* [6]). 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 5.1 and 5.2.

Lemma 5.3 *Every nice execution of our algorithm satisfies requirement 5.1 and requirement 5.2.*

Proof: We first prove that requirement 5.1 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 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 E then the agent must arrive at p_i following the delivery of m and before it is removed.

We now show that requirement 5.2 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 E before m_1 is sent and let p_i and p_j be two processors that delivers m_0. Every processor p_i that receives an agent and delivers m_0, either finds m_1 as well in the agent (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 been sent yet) 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 non-empty 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., [24, 28, 7, 11, 13]). In [29] 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 [29] is for an advanced resource allocation task, but is not self-stabilizing. Here in contrast we present a self-stabilizing solution for a basic resource allocation task.

Resource Allocation Requirement:

Requirement 6.1 *Let E be a nice execution, such that every processor that possesses the resource releases it after B asynchronous cycles, for some finite constant B. Then every processor p_i that wishes to possess the resource infinitely often, possesses the resource infinitely often.*

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 [6].

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 non-primary and are limited to executing a reduced set of operations [6].

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 (maybe the only active processors) forms a primary view. Consider an execution E_1 in which A is a primary view that consists of a set of processors, and consider a different execution E_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 E_3 with two disjoint "primary" connected components A and A' in the first configuration of an execution E_3. Since there is no communication between the two components, then 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 6.2 *Every nice execution of our algorithm satisfies requirement 6.1.*

Proof: Let p_i a processor that wishes to possess the resource in E. 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 r, 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., [4, 25]) 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.

Acknowledgment: Many thanks to Uriel Feige for helpful discussions and Lyn Pierce for improving the presentation.

References

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

[2] Y. Amir, L. Moser, D. Agrawal, and P. Ciarfella, "Fast message ordering and membership using a logical token-passing ring", *Proc. 13th IEEE International Conference on Distributed Computing Systems*, Pittsburgh, PA, pp. 551–560, 1993.

[3] J. Beauquier, T. Herault, and E. Schiller, "Easy Self-stabilization with an Agent" *5th Workshop on Self-Stabilizing Systems*, Lisbon, Portugal pp. 35–50, 2001.

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

[5] I. Chatzigiannakis, S. Nikoletseas, and P. Spirakis, "An Efficient Communication Strategy for Ad-Hoc Mobile Networks", *Proc. 15th International Symposium on Distributed Computing*, Lisbon, Portugal, pp. 285-299, 2001.

[6] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group Communication Specifications: A Comprehensive Study", *ACM Computing Surveys*, 33(4):1–43, December 2001.

[7] M. Choy and A. K. Singh, "Efficient fault tolerant algorithms for distributed resource allocation," *ACM Transactions on Programming Languages and Systems*, 17(4):535–559, 1995.

[8] D. Coppersmith, P. Tetali, and P. Winkler, "Collisions among Random Walks on a Graph," *SIAM J. on Discrete Math.*, 6(3):363–374, August 1993.

[9] 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 at San Diego, 1995.

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

[11] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, 1:115–138, 1971.

[12] E. W. Dijkstra, "Self stabilizing systems in spite of distributed control," *Communications of the ACM*, 17:643–644, 1974.

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

[14] S. Dolev and E. Schiller, "Communication Adaptive Self-Stabilizing Group Membership Service," *Proc. 5th Workshop on Self-Stabilization, WSS'01*, LNCS 2194, pp. 81–97, 2001 (also BGU TR-02, July 2000).

[15] U. Feige, "A Tight Upper Bound on the Cover Time for Random Walks on Graphs," *Random Structures and Algorithms*, 6(4):51–54, 1995.

[16] U. Feige, "A Tight Lower Bound on the Cover Time for Random Walks on Graphs," *Random Structures and Algorithms*, 6(4):433–438, 1995.

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

[18] A. Fekete, N. Lynch, and A. Shvartsman, "Specifying and Using a Partitionable Group Communication Service," *Proc. 16th Annual ACM Symposium on Principles of Distributed Computing*, Santa Barbara, CA, pp. 53–71, 1997.

[19] S. Ghosh, "Agents, distributed algorithms, and stabilization," *Computing and Combinatorics*, Springer-Verlag LNCS:1858, 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 Symposium on Parallel Algorithms and Architectures*, pp. 251–260, 1999.

[21] T. Herman and T. Masuzawa, "Self-stabilizing Agent Traversal," *Proc. 5th Workshop on 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.

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

[24] N.A. Lynch, "Fast allocation of nearby resources in a distributed system," *Proc. 12th ACM Symposium on Theory of Computing*, pp. 70–81, 1980.

[25] G. Neiger, "A New Look at Membership Service," *Proc. 15th ACM Symposium on Principles of Distributed Computing*, 1996.

[26] V. D. Park and M. S. Corson, "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks," *Proc. IEEE INFOCOM*, April 1997.

[27] B. Rajagopalan and P. McKinley, "A Token-Based Protocol for Reliable, Ordered Multicast Communication," *Proc. 8th IEEE Symposium on Reliable Distributed Systems*, Seattle, WA, pp. 84–93, October 1989.

[28] E. Steyer and G. Peterson, "Improved Algorithms for Distributed Resource Allocation," *Proc. 7th Symposium on Principles of Distributed Computing*, pp. 105–116, 1988.

[29] J. Sussman, and K. Marzullo, "The Bancomat Problem: An Example of Resource Allocation in a Partitionable Asynchronous System," *Proc. 12th International Symposium on Distributed Computing (DISC)*, 1998.

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