Autonomous Virtual Mobile Nodes

(Extended Abstract)

Shlomi Dolev* Seth Gilbert† Elad Schiller*‡ Alex Shvartsman§† Jennifer Welch¶

June 3, 2005

Abstract

This paper presents a new abstraction for virtual infrastructure in mobile ad hoc networks. An Autonomous Virtual Mobile Node (AVMN) is a robust and reliable entity that is designed to cope with the inherent difficulties caused by processors arriving, leaving, and moving according to their own agendas, as well as with failures and energy limitations. There are many types of applications that may make use of the AVMN infrastructure: tracking, supporting mobile users, or searching for energy sources.

The AVMN extends the focal point abstraction in [9] and the virtual mobile node abstraction in [10]. The new abstraction is that of a virtual general-purpose computing entity, an automaton that can make autonomous on-line decisions concerning its own movement. We describe a self-stabilizing implementation of this new abstraction that is resilient to the chaotic behavior of the physical processors and provides automatic recovery from any corrupted state of the system.

1 Introduction

Ad hoc infrastructure for mobile ad hoc networks is desperately needed to make these systems usable by applications, allowing developers to overcome the numerous inherent difficulties, such as processors arriving, leaving and moving according to their own agendas, as well as by failures and energy limitations.

This paper introduces a new abstraction that extends the focal point abstraction in [9] and the virtual mobile node abstraction in [10]. The new abstraction is that of a virtual general-purpose computing entity, an automaton that can make autonomous on-line decisions concerning its own movement. We call this abstraction an Autonomous Virtual Mobile Node (AVMN). We describe an implementation of this new abstraction that is resilient to the chaotic behavior of the underlying network. Moreover, it guarantees automatic recovery from any corrupted system state.

*Ben-Gurion University, {dolev,schiller}@cs.bgu.ac.il
†MIT CSAIL, {sethg, alex}@theory.lcs.mit.edu
‡Research Academic Computer Technology Institute, schiller@cti.gr
§Department of Computer Science and Engineering, University of Connecticut, aas@cse.uconn.edu
¶Texas A&M University, welch@cs.tamu.edu

[1]This work is supported in part by NSF grant CCR-0098305 and NSF ITR Grant 0121277. The first author and third authors are partially supported by an IBM faculty award, the Israeli ministry of defense, NSF, and the Rita Altura trust chair in computer sciences. The second author is partially supported by AFOSR Contract #F49620-00-1-0097, DARPA Contract #F33615-01-C-1896, NSF Grant 64961-CS, NTT Grant MIT9904-12. The fourth author is partially supported by the NSF Grant 9988304, 0311368 and by the NSF CAREER Award 9984774. The fifth author is partially supported by NSF Grant 0098305.

At any given point in time, the AVMN resides at a distinct location. The AVMN is implemented by the processors that happen to be near the AVMN's current location, thus enhancing the robustness as processors fail and move out of range. The set of processors implementing the AVMN changes over time as the AVMN moves and as the implementing processors move (not necessarily in the same direction). Despite the continually changing set of participants, from a client's perspective, the AVMN acts like a single, monolithic entity.

One of the primary differences between an AVMN, introduced in this paper, and a virtual mobile node (see [10]) is that an AVMN can move autonomously, choosing to move based on its current state and sensor inputs from the physical environment. For instance, if the area to the west of the AVMN appears deserted, then it may decide not to move west. On the other hand, the AVMN may decide to "hitch a ride" with a subset of the processors currently emulating it. In contrast, the virtual mobile node was required to fix a predetermined path in advance, when the algorithm was deployed, thus significantly limiting the flexibility of the virtual node.

Allowing the AVMN to move autonomously introduces several challenges. First, the algorithm must ensure that a consistent set of processors is used to implement the AVMN. When an AVMN decides to move, however, the set of processors participating in the emulation may change; in transitioning from the old set of processors to the new set of processors, the emulator must ensure an orderly transition while maintaining consistency and liveness. The second problem introduced by autonomy is the lack of *a priori* location at which the AVMN can be found. Therefore, when the AVMN fails (e.g., due to entering an empty region of the network where there are no processors to participate in the emulation), it can be quite difficult to detect this failure and restore the AVMN.

Our AVMN implementation is also self-stabilizing, in that it can tolerate the processors' starting from an arbitrary configuration. If a state corruption causes two different sets of processors to begin emulating the same AVMN, the emulation algorithm detects this situation and corrects it. Moreover, if the emulating processors become inconsistent (for example, due to network abnormalities), the emulator can recover from the state corruption, and continue to operate correctly.

Roadmap. In the rest of this section, we discuss prior work, in particular focusing on virtual infrastructures in wireless ad hoc networks. In Section 2, we present the underlying model for wireless ad hoc networks. In Section 3, we define the required properties of an AVMN in more detail. We then proceed to present a self-stabilizing algorithm to emulate an AVMN. Our implementation consists of two parts. The first part, a basic emulator that operates correctly once the set of participants is consistent, is presented in Section 4. The second part ensures that the set of participants eventually stabilizes to a consistent set, and is presented in Section 5. We present some discussion and optimizations in Section 6.

Previous work. In [9], we presented a new approach, called GeoQuorums, for implementing atomic read-/write shared registers in mobile *ad hoc* networks. This approach is based on associating abstract atomic objects with certain geographic locations called "focal points". These geographic locations are assumed to be normally populated by mobile processors. In [10], we generalized our approach from [9] from stationary atomic objects to mobile virtual nodes. We assumed that the virtual node moves on a fixed trajectory that is globally known in advance. We presented a new replicated state machine algorithm to implement the virtual node using a constantly changing set of processors in the vicinity of the virtual node's current location.

In contrast with [10], our current work relaxes the assumption that the trajectory of each virtual entity is fixed and known in advance. Furthermore, the new abstraction is self-stabilizing and automatically regenerating. Fixed-location self-stabilizing virtual stationary automata for different settings appear in [11, 8]. As discussed above, the introduction of autonomy introduces several new difficulties.

The idea of executing algorithms on virtual mobile entities was inspired by compulsory protocols [15, 6, 19], which assume that some subset of the processors can control their own motion. They showed that this assumption significantly simplifies the design of protocols, compared to an environment in which processors move in an unpredictable or adversarial manner. The work in [10] on virtual mobile nodes generalizes Beal's Persistent Node abstraction [1, 2], in which nodes travel in a static network carrying limited state. The work of Nath and Niculescu [22], in which messages are routed along a particular trajectory, and Geocast (e.g., [23, 5, 17]), in which data is routed geographically, are connected to this work in that they can be seen as attempts to simulate a traveling processor with limited functionality.

2 Basic System Model

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

The processors communicate among themselves using a local broadcast primitive, with radius R_{lb}. The local broadcast is assumed to be reliable, meaning that every processor that stays within distance R_{lb} of the sending processor is guaranteed to receive the message exactly once, and to ensure delivery within d time. This is an abstraction of some Ethernet-like service. The operations are denoted LBcast and LBrecv.

There is a Geocast service, by which a processor can send a message to all processors in some specified geographic area. We also assume the Geocast is reliable and that there is an upper bound $D \gg d$ on the latency of Geocast messages. A number of Geocast routing protocols have been proposed for mobile ad hoc networks (see [25] for a survey and comparison). The operations are denoted Geocast and Georecv.

Finally, we assume that there is a reliable time and location service available to each processor, such as would be provided by GPS. The existence of a reliable time and location service makes it easy to implement the local broadcast and Geocast communication services in a self-stabilizing way, by differentiating current messages from previous (possibly corrupted) messages.

Several *processes* can run in a single processor. The inputs to a process include the receipt of a message destined for itself, either from another processor or from the same processor. For instance, there could be a process associated with a sensor on the processor that sends data to another process on the same processor. 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. Each step of p_i is triggered by an input, which is either the receipt of a message or a timer going off. 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. The execution of a step in the algorithm can change the state of a processor.

We let the undirected graph $G(\mathcal{V}, \mathcal{E})$ denote the current communication graph of the system, where $\mathcal{V}$ is the set of processors, together with their coordinates in the plane, and there is an edge in $\mathcal{E}$ between processors p_i and p_j if and only if the two processors can communicate with each other. (This depends on whether the two processors are within R_{lb} of each other). Notice that G changes over time.

The term *system configuration* is used for a tuple of the form $(s_1, s_2, \cdots, s_n, G(\mathcal{V}, \mathcal{E}))$, where each s_i is the state of processor p_i (including messages in transit for p_i)and $G(\mathcal{V}, \mathcal{E})$ is the current communication topology. Therefore the vector of individual processor states and the current communication graph fully describes the system state.

We define an *execution* $E = c_0, st_0, c_1, st_1, \ldots$ as an alternating sequence of system configurations c_i and steps st_i, such that each configuration c_{i+1} (except the initial configuration c_0) is obtained from the preceding configuration c_i by the execution of the step st_i. In addition, st_i may reflect a change in the communication graph. Thus, the only components that can be changed due to the execution of st_i are the state of p, the state of a neighbor of p and the communication graph $G(\mathcal{V}, \mathcal{E})$. An execution is *fair* if every processor executes a step infinitely often.

In some of our algorithms, random walks are used for broadcasting information. We consider the subset of fair executions in which a message sent in a random walk fashion succeeds in arriving at all processors in the system in a timely fashion. A *nice execution* is defined [12] to be an execution in which a message sent in a random walk fashion arrives at every processor in at most every M consecutive message send operations, where M is a constant that depends on n. The probability of having a nice execution in several common cases is computed in [12] using techniques from random walks. (See, for example, [20] for standard calculations of cover times in various graphs). The probability is calculated assuming an arbitrary initial configuration and relies on known results about the cover time of random walks in graphs. For our algorithms that use random walks, we prove that every nice execution satisfies the desired conditions (defined as the requirements below).

3 Autonomous Virtual Mobile Nodes

An Autonomous Virtual Mobile Node (AVMN) is an arbitrary automaton that resides, at any given time, at a specific location in the network; it can communicate with nearby processors, using the local broadcast service, and send and receive Geocast messages in the same way as a real processor residing at its location. The AVMN is specified in terms of (1) a set of states, V, (2) an initial statue, v_0, (3) a set of inputs, `inputs`, (4) a set of outputs, `outputs`, and (5) a transition function, δ, mapping from states and inputs to states and outputs. An algorithm implementing an AVMN must satisfy the following property:

Property 1 (Correct emulation of the AVMN) *The execution of the AVMN implementation produces an external trace that is consistent with a state change sequence that is correct according to the transition function, δ, of the AVMN.*

Unlike a processor, an AVMN controls its own motion: an AVMN moves in discrete steps from one location to another. An AVMN specification, then, also includes a movement function, $\mathrm{calculate-location}$, which determines a new location for the AVMN as a function of its current location and current state.

Finally, AVMNs are robust. As long as there are real processors near the AVMN, it remains alive. There are two ways an AVMN can fail: either it enters an empty region of the network, or it suffers a state corruption, potentially causing multiple copies of the AVMN to appear in the network. In either case, it can recover.

Property 2 (Exactly one AVMN location) *Eventually there is exactly one copy of an AVMN in the network.*

An AVMN is *self-stabilizing*, in that in every fair/nice execution that starts in an arbitrary configuration there is a suffix in which Properties 1 and 2 are satisfied.

The program (including the AVMN code) of the processors is assumed to be (hardwired and) correct, namely, we do not assume Byzantine behavior of the processors. Note that an AVMN-simulation process needs to be running all the time, even if just listening to messages to see if it should start participating.

We also assume that the program consists of information concerning N, the upper bound on the number of processors and the identifier of the processor.

We remark that the application that uses the AVMN as a computing platform should be self-stabilizing as well, since the AVMN may start correct execution of the application from an arbitrary state.

4 Self-Stabilizing Implementation of an AVMN

In this section we describe the basic algorithm to emulate an AVMN, assuming all the participants in the emulation are near, within some fixed $R_{avmn} < R_{lb}$ of, the unique location of the AVMN, that is, if the AVMN has a *consistent set* of participants. In Section 5, we show how to ensure that there is a consistent set of participants. The pseudocode for the basic AVMN emulator appears in Figure 1 (and all line numbers refer to this figure).

Replication. Each participating processor keeps a replica of the AVMN's current state and a buffer of input events waiting to be applied to the state. It is sufficient to keep only the events that have occurred within the last $2d$ time units, where d is an upper bound on the latency of the local broadcast service.

The emulation protocol must ensure that state transitions of the AVMN are atomic and identical in all replicas. A state transition can be triggered by inputs, such as the messages arriving (via Geocast) at a participating processor, sensor inputs, or the clock reaching a certain value. When a processor receives a Geocast message or detects a sensor input, it broadcasts a message using the LBcast service indicating that an event occurred (lines 27 and 30). On receiving a message (lines 19–24), an additional delay of d, the maximum broadcast delay, is imposed (via a timer—line 24) to ensure that all processors process the events in the same order. This ensures that the state is updated consistently.

To ensure that the replica states remain identical among all the processors that emulate the AVMN, in spite of faults, each processor, at a fixed interval, sends its replica state (or a hash function thereof) to all the other emulating processors (lines 32–35). Upon receiving all the messages, at least d time after the time at which the checkpoints were sent, a processor checks if there are any conflicts, that is, the states received are not identical (line 64). In this case, a predetermined recovery function is applied (line 65), and the buffers are flushed (lines 67–72).

Joining. When a processor enters the "sphere of influence" of an AVMN, that is, within R_{avmn}, it should start participating in the simulation of the AVMN (lines 77–84). The joining processor sets is status to joining, and waits for a state refresh. During this time, it listens, saving the events in its buffer. After d time passes, it has the same buffer as all other actively participating processors. Therefore, the first time the processor receives a state refresh that was initiated at least d time after it began listening, it can complete the join protocol by adopting the new state (lines 73–75). (Note that in an optimized version where only a hash is sent, the joining processor will have to request the state explicitly.)

Suppose, as in Figure 2, the joiner starts the join procedure at time t (setting its own $\texttt{last}-\texttt{refresh}$ to t). The joiner takes the first replica state that it receives with timestamp (i.e., $\texttt{lr}$) at least $t+d$. Call this timestamp t'. It collects all the replica states with timestamp t', checking for consistency. The joiner then adopts this state and replays all messages that it has received with timestamp greater than $lr-d$ using the usual delivery algorithm, processing the messages in order of their timestamp, ignoring message sent in the last d time and breaking ties in some consistent way.

Navigation. A key feature of the AVMN is that it can decide autonomously where to move. The decision is a function of the current state of the AVMN, which may encode information concerning the current

Figure 1: AVMN Emulator

```
Variables:
  status, in {idle, joining, active}
  state, state of the replica
  location, current VMN location
  buffer, buffer for incoming messages
  last−refresh, last time a state refresh occurred
  clock, real time clock

Externally specified functions/constants:
  v0, the initial state of the AVMN
  δ, the AVMN transition function
  calculate−location(...), calculates the next location of the AVMN
  recover(...), deterministically chooses a new state from a set of old states
  t_move, frequency of movement
  t_refresh, a state refresh intervale
  t_process, speed at which AVMN takes spontaneous steps
```

```
init(ℓ)
  location ← ℓ
  state ← v0
  buffer ← ∅
  last−refresh ← clock
  status ← active
  settimer(next−multiple(t_refresh),Refresh)
  settimer(next−multiple(t_process),Process)
  settimer(next−multiple(t_move),Move)

LBrecv(m)
  if (m = ⟨new−loc,ℓ⟩) and (status = idle) then
    location ← ℓ
  else
    buffer ← buffer ∪ ⟨m,clock⟩
    settimer(clock+d,NewMessage)

Georecv(m)
  LBcast(⟨sim,Georecv(m)⟩)

onSensor(m)
  LBcast(⟨sim,Sensor(m)⟩)

onTimer(RefreshState)
  LBcast(⟨state,state,clock⟩)
  last−refresh ← clock
  settimer(next−multiple(t_refresh),RefreshState)

onTimer(Process)
  if ∃ x : δ(state,Geocast(x)) ≠ ⊥ then
    lbcast(geo,x)
  if ∀⟨m,t⟩ ∈ buffer : t < clock−2d then
    buffer ← buffer \ ⟨m,t⟩
  settimer(next−multiple(t_process),Process)

onTimer(Move)
  LBcast(move,location,clock)
  settimer(next−multiple(t_move),Move)

onTimer(NewMessage)
  let m = min(m : ⟨m,t⟩ ∈ buffer, t = clock−d)
  if (m = ⟨new−loc,ℓ⟩) then
    location ← ℓ
  if (status = active) then
    if m = ⟨sim,x⟩ then
      states ← δ(state, m)
    else if m = ⟨geo,x⟩ then
      states ← δ(state, m)
      Geocast(x)
    else if m = ⟨move,loc,move−time⟩ then
      if (loc = location) then
        location ← calculate−location(location, state)
        LBcast(new−loc,ℓ)
    else if m = ⟨state,x,lr⟩ then
      let S = {m : m = ⟨state,y,lr⟩}
      if (|S| > 1) or (status = joining) then
        state ← recover(S)
        let J = {⟨m,t⟩ ∈ buffer : lr−d ≤ t ≤ clock−d}
        while J ≠ ∅
          let m′ = min(J)
          if m′ = ⟨sim,y⟩ then
            states ← δ(state,y)
            J ← J \ m′
            buffer ← buffer \ m′
          if (status = joining) and (last−refresh+d ≤ lr) then
            status ← active
            settimer(next−multiple(t_refresh),RefreshState)

onNewLocaction(ℓ)
  if (|ℓ −location|) < R then
    if (status = idle) then
      status ← joining
      last−refresh ← clock
      cleartimers()
  else
    status ← idle
```

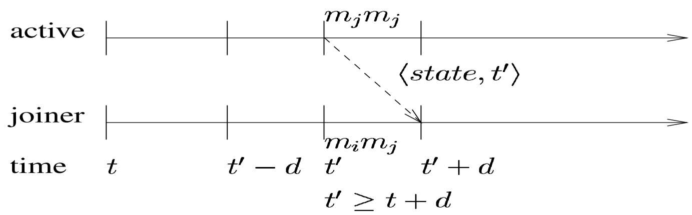

Figure 2: The joiner adopts state received at time $t' + d$, quickly replays the m_i, m_j messages, and then is caught up. Note that in the figure, the m_i, m_j messages are *sent* in the interval $[t' - d, t']$ and delivered in $[t', t' + d]$.

environment. With a fixed frequency, t_{move}, a processor participating in the emulation initiates a move (lines 44–46). Notice that this broadcast message does not actually specify the location, as might be expected. In fact, each processor independently calculates the new location, based on the old location, the time of the move, and the current state (line 60). The primary purpose of this broadcast message is to order the movement with respect to the other messages and events being processed, in order to ensure that the move occurs consistently at all processors. As a result, when the new location is calculated all the processors have the same replicated state, and therefore choose the same new location.

After the new location is calculated, a broadcast message is sent notifying all the processors of the new AVMN location (line 61). Only participating processors can calculate the new location themselves; other processors that are not participating receive the $new - loc$ message, updating them on the current location. Without this additional message, no new nodes would be aware of the new location and would be unable to join the emulation.

In order that enough old nodes remain participants, and that enough nodes near the new location can receive the notification, we impose an additional limitation on the speed of motion. Let ϵ be the maximum distance moved by the AVMN in a single transition. Then we assume that $R_{lb} \geq 2 \cdot R_{avmn} + \epsilon$.

Theorem 3 *If at some point in the execution there is a consistent set of participants, then from that point on the trace is consistent with a state change sequence that is correct according to the input and transition functions.*

Proof. (*sketch*) First, notice that every participating processor that is with R_{avmn} of the AVMN location processes messages in the same order. That is, there exists a total ordering of all messages, based on the time they were sent; every processor removes them from the buffer in that order: before processing a message, m, a processor delays d time, therefore by the time m is removed from the buffer, every message sent prior to m has been received.

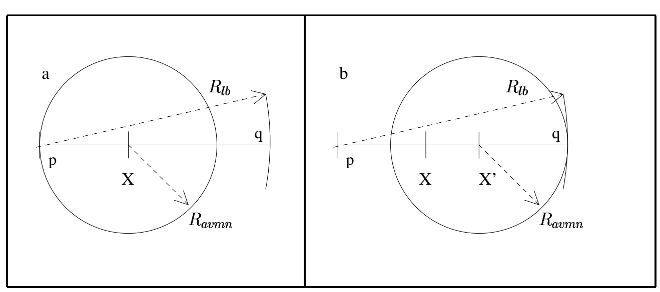

Figure 3: a) Processor p participates in a AVMN at location X and informs processor q about relocation to X' using an R_{lb} broadcast. b) Processor q participates in a AVMN at new location X' that is ϵ distance units away.

The proof then follows by induction on the sequence of messages processed. The following two invariants are maintained: (1) all processors have the same replica state after processing message m_k, (2) the set of participating processors is consistent. This follows by a case analysis of the messages processed. If m_k is a sim message or a geo message, then the state is consistently updated at all processors by applying m to the current state, which by induction and consistent message ordering is the same at all nodes. If m_k is a state message, then either the states are already consistent, or recovery begins. In the latter case, each processor has the same set of state messages in its buffer, and the same set of other old messages, and so chooses the same new state. (Note this also shows that joining is successful.) If m_k is a move message, then each processor that receives the message is either still near the new center, in which case it remains a participant, or it is far from the new center, in which case it leaves. If m_k is a $\mathrm{new-loc}$ message and p_i is not active, it simply adopts the new location; since it was previously not a participant, the set of participants is still consistent.

At this point, it is easy to show that the trace is consistent with an AVMN execution, based on the states after each message is received. Hence, any externally visible action, such as a Geocast, is consistent. ■

5 Ensuring Existence of Exactly One AVMN

Recall that Theorem 3 guarantee a consistent execution from that point at which there is a consistent set of participants. In this section, we describe how to stabilize on a consistent set of processors to emulate the AVMN, presenting three schemes for ensuring the existence of exactly one instance of an AVMN.

Virtual Stationary Automata Scheme. The first scheme uses a virtual stationary automaton (VSA) to keep track of the AVMN. A VSA is another type of virtual infrastructure component, introduced in [11]. Unlike an AVMN, it is stationary, fixed in a single predetermined location. Much like an AVMN, it is emulated by a set of continually changing participants. Since it is stationary, however, the issues of autonomy do not arise. In particular, for a VSA it is trivial to ensure a consistent set of participants: they are exactly the set of participants that are near the VSA's fixed location. One could therefore implement a VSA using the algorithm in Section 3, instead of the algorithm in [11].

A VSA, if available, can be used to simplify the problem of maintaining a consistent set of participants in an AVMN. The AVMN uses a Geocast service to send "I am alive" messages to the region containing the VSA. If the VSA does not receive an "I am alive" message for too long a period, the VSA creates a new AVMN. The VSA is also responsible for the elimination of undesired copies of an AVMN. Each "I am alive" message carries the location of the AVMN and the timestamp at which the message was sent. The VSA can easily detect that more than one copy of the AVMN exists and send an elimination message to all but one of them. The scheme can be naturally extended to a more fault tolerant, distributed version in which several VSAs are responsible for the existence of the AVMN, each having a different time-out period to avoid simultaneous creation of multiple copies.

Lemma 4 *Starting from an arbitrary initial state, the VSA Scheme ensures a consistent set of participants.*

Proof. (*sketch*) If there is no AVMN in the network, then eventually the VSA stops receiving "I am alive" messages and creates a unique new one. If there is more than one AVMN in the network, then eventually the VSA eliminates all but one. ■

We note that, in the VSA Scheme, starting from an arbitrary configuration, we reach a consistent set of participants within: (1) the time it take the VSA to stabilize, plus (2) the Geocast time.

Token Random Walk Scheme. In the second scheme, the mobile processors themselves verify the existence of the AVMN, without relying on an auxiliary VSA. The AVMN repeatedly sends out a token containing the message "I am alive." The token travels on a random walk through the ad hoc network, until its time-to-live expires. If a processor does not receive an "I am alive" token for, say twice, the expected random walk cover time (see [20, 12], for example, for cover time bounds), then it generates a token containing a "formation" message and the processor's identifier and a time-to-live that bounds the token's lifetime. The formation token itself travels on a random walk. When two formation tokens collide, they merge, maintaining a collection of processor identifiers. When a formation token contains $\lceil (N+1)/2 \rceil$ processor identifiers, the (single) processor that holds the token creates a new AVMN.

To ensure that there is eventually only one copy of the AVMN, each AVMN monitors the "I am alive" messages in the network, each of which includes a timestamp and a location. The AVMN, which maintains a bounded location history, can thus determine if a token belongs to a duplicate AVMN, and determine using a deterministic function whether to eliminate itself. Only a bounded history is needed since there exist bounds on how long it takes a token to cover the network in nice executions.

Lemma 5 *Starting from an arbitrary initial state, the Token Random Walk Scheme ensures a consistent set of participants.*

Proof. (*sketch*) If there is no AVMN in the system, eventually each processor produces a formation token. Eventually, the formation tokens collide, forming a unique AVMN. If there is more than one AVMN, eventually each AVMN receives "I am alive" tokens from the other AVMNs. All but one AVMN will then be eliminated. ∎

We note that, in the Token Random Walk Scheme, starting from an arbitrary configuration, we reach a consistent set of participants within order M time, where M is the time it takes for a random walk to visit every node (see [12]).

Stay Alive Scheme. The third scheme is different in the sense that the AVMN itself does not send messages. Instead, processors at predefined times (say every hour on the hour) send tokens containing a "stay alive" message on a random walk of the network. Eventually the AVMN should receive the tokens. In every time period the AVMN must collect at least $\lceil (N+1)/2 \rceil$ stay alive tokens in order to survive the next time period. Notice that if there is more than one copy of the AVMN, at most one is able to collect a majority of stay alive tokens in a time period. If a stay alive token survives for too long without finding an AVMN, it begins to act like a formation token in the Token Random Walk scheme: when two stay alive formation tokens collide, they merge, and when a majority of stay alive formation tokens have merged, they form a new AVMN.

Lemma 6 *Starting from an arbitrary initial state, the Stay Alive Scheme ensures a consistent set of participants.*

Proof. (*sketch*) If there is no AVMN in the system, eventually the tokens all become formation tokens, and eventually all merge and form a new AVMN. If there is more than on AVMN in the system, at most one is able to collect a majority of the tokens, and therefore at most one AVMN survives. ∎

As in the Token Random Walk Scheme, in the Stay Alive Scheme, when starting from an arbitrary configuration, we reach a consistent set of participants within order M time, where M is the time for a random walk to visit every processor.

Trade-Offs. The VSA scheme is the most efficient, in terms of messages required. Unlike the other two schemes, messages can be sent directly to a known location, rather than performing a random walk of the network. For the same reason, the VSA scheme is able to respond most rapidly to abnormalities in the system. In fact, the simplicity of this scheme is yet another example of the utility of having virtual, reliable infrastructure in a mobile ad hoc network.

On the other hand, the VSA scheme requires maintaining a stationary virtual automaton. The Token Random Walk scheme is also relatively message efficient, in that in the stable state when there exists one AVMN, there only needs to be a small number of tokens performing random walks in the network. It is only in the case of formation that all the processors need to create tokens.

The Stay Alive scheme is the least efficient, in terms of messages. All the processors need to create tokens at all times. However, it is simpler than the Token Random Walk scheme, in that only one type of token is needed. Moreover, the AVMN does not have to send any heartbeat messages.

Using any of the three schemes, we can conclude our main theorem:

Theorem 7 *The AVMN emulator, using any of the three schemes, is a self-stabilizing implementation of an arbitrary automaton.*

6 Discussion

We have discussed how to implement a single AVMN; one could instead have multiple AVMNs, possibly dynamically created by AVMN cloning, performing different tasks, and collaborating among themselves. Moreover, AVMNs can be organized into a hierarchy, and be used as robust entities for tracking, updating, communicating and more, in the scope UAVs, sensor networks, ad-hoc networks, and RFID tags (for instance, the AVMN may follow the energy source/beam of light).

There are a number of ways to optimize the movement of the AVMN so as to minimize the energy needed during a broadcast. First, the processor can examine its current location and only use the minimum amount of power necessary to reach everyone at the new AVMN location. Second, we can use the mobile processors that are closer to the new AVMN location to perform the broadcast. Hence, only these need to use more broadcast power. Third, if the AVMN motion can be dependent on the mobile processor's motion (for example, in the case of tracking), then we can take advantage of the movement of the mobile processors to minimize the energy needed. In some application domains, the AVMN is allowed decide to start controlling the movement of the mobile nodes that implement it (e.g., [7, 13, 14, 16]).

The algorithm presented can be optimized in many ways, for example, the communication overhead can be significantly reduced by using checksums (instead of sending the entire state) and/or using randomization to limit the number of processors broadcasting consistency-check messages. When an inconsistency is detected, we can use an ethernet-like algorithm to choose randomly which replica will survive (it will be the first that succeeds in performing local broadcast).

We also note that there are ways to change the AVMN program that is assumed to be hardwired in each processor. One way to do so is by using a super-user message that is sent to all the processors (say, with the assistance of VSAs) to replace their code.

Our approach can also be generalized to work in three dimensions, rather than two — instead of a disc around the AVMNs location, we may consider a ball.

References

[1] J. Beal, “Persistent nodes for reliable memory in geographically local networks,” TR AIM-2003-11, MIT, 2003.

[2] J. Beal, “A robust amorphous hierarchy from persistent nodes,” *Proc. of Communication Systems and Networks*, 2003.

[3] O. B. Bayazit, J.-M. Lien, and N. M. Amato, “Roadmap-Based Flocking for Complex Environments,” *Proc. 10th Pacific Conference on Computer Graphics and Applications (PG’02)*, 2002.

[4] J. Bohn and F. Mattern, “Super-Distributed RFID Tag Infrastructers,” TR, Institute of Pervasive Computing, ETH, 2004.

[5] T. Camp and Y. Liu, “An adaptive mesh-based protocol for geocast routing,” *Journal of Parallel and Distributed Computing: Special Issue on Mobile Ad-hoc Networking and Computing*, pp. 196–213, 2002.

[6] I. Chatzigiannakis, S. Nikoletseas, and P. Spirakis, “An efficient communication strategy for ad-hoc mobile networks,” *Proc. 15th International Symposium on Distributed Computing*, 2001.

[7] P. Chandler and M. Pachter, “Hierarchical Control for Autonomous Teams”, *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2001.

[8] S. Dolev and O. Gersten, “Robust Active Super Tier Systems”, *Proc. of the IEEE International Conference on Software-Science, Technology and & Engineering*, 2005.

[9] S. Dolev, S. Gilbert, N. Lynch, A. Shvartsman, and J. L. Welch, “GeoQuorums: Implementing Atomic Memory in Ad Hoc Networks”, *Proc. 17th International Symposium on Distributed Computing (DISC)*, pp. 306–320, 2003. To appear in *Distributed Computing*.

[10] S. Dolev, S. Gilbert, N. Lynch, E. Schiller, A. Shvartsman, and J. L. Welch, “Virtual Mobile Nodes for Mobile Ad Hoc Networks,” *Proc. 18th International Symposium on Distributed Computing (DISC)*, pp. 230–244, 2004.

[11] S. Dolev, S. Gilbert, L. Lahiani, N. Lynch, and T. Nolte, “Virtual Stationary Automata for Mobile Networks”, TR MIT-LCS-TR-979, MIT CSAIL, Cambridge, MA 02139, January 2005.

[12] S. Dolev, E. Schiller, and J. L. Welch, “Random Walk for Self-Stabilizing Group Communication in Ad-Hoc Networks,” *Proc. 21st Symp. on Reliable Distributed Systems,* pp. 70–79, 2002. To appear in *IEEE Transactions on Mobile Computing*.

[13] D. Gillen and D. Jaques, “Cooperative Behavior Schemes for Improving the Effectiveness of Autonomous Wide Area Search Munitions”, *Proceedings of the Cooperative Control Workshop*, 2000.

[14] J. Hebert, “Cooperative Control of UAVs”, *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2001.

[15] K. P. Hatzis, G. P. Pentaris, P. G. Spirakis, V. T. Tampakas, and R. B. Tan, “Fundamental control algorithms in mobile networks,” *Proc. of the 11th ACM Symposium on Parallel Algorithms and Architectures archive*, Saint Malo, France, 1999.

[16] E. Kivelevich and P. Gurfil “UAV Flock Taxonomy and Mission Execution Performance”, *Proc. of the 45th Israeli Conference on Aerospace Sciences*, 2005.

[17] F. Kuhn, R. Wattenhofer, Y. Zhang, and A. Zollinger., “Geometric Ad-Hoc Routing: Of Theory and Practice”, *Proc. of the 22nd Symp. on the Principles of Distributed Computing*, July 2003.

[18] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, 21(7):558–565, 1978.

[19] Q. Li and D. Rus, “Sending messages to mobile users in disconnected ad-hoc wireless networks,” *Proc. 6th MobiCom*, 2000.

[20] R. Motwani and P.Raghavan, “Randomized Algorithms.” Cambridge University Press, 1995.

[21] R. Nagpal, H. Shrobe, and J. Bachrach, “Organizing a global coordinate system from local information on an ad hoc sensor network,” *2nd Workshop on Information Processing in Sensor Networks*, 2003.

[22] B. Nath and D. Niculescu, “Routing on a curve,” *ACM SIGCOMM Computer Communication Review*, 33(1):150 – 160, 2003.

[23] J. C. Navas and T. Imielinski. “Geocast – geographic addressing and routing,” *Proc. of the 3rd MobiCom*, 1997.

[24] N. B. Priyantha, A. Chakraborty, H. Balakrishnan. “The cricket location-support system,” *Proc. 6th ACM MOBICOM*, 2000.

[25] P. Yao, E. Krohne, and T. Camp, “Performance Comparison of Geocast Routing Protocols for a MANET,” *Proc. of the 13th IEEE International Conference on Computer Communications and Networks (IC3N),* pp. 213–220, 2004.