Virtual Stationary Automata for Mobile Networks

Shlomi Dolev,* Seth Gilbert,† Limor Lahiani,* Nancy Lynch,† Tina Nolte†

***Abstract*— We define a programming abstraction for mobile networks called the *Virtual Stationary Automata* programming layer, consisting of real mobile clients, virtual timed I/O automata called virtual stationary automata (VSAs), and a communication service connecting VSAs and client nodes. The VSAs are located at prespecified *regions* that tile the plane, defining a static virtual infrastructure. We present a self-stabilizing algorithm to emulate a VSA using the real mobile nodes that are currently residing in the VSA's region. We also describe several examples of applications whose implementations benefit from the simplicity obtained through use of the VSA abstraction.**

I. Introduction

The task of designing algorithms for constantly changing networks is difficult. Highly dynamic networks, however, are becoming increasingly prevalent, especially in the context of pervasive and ubiquitous computing, and it is therefore important to develop new techniques to simplify this task.

In this paper we focus on mobile ad-hoc networks, where mobile processors wander the world, coordinating their computation despite minimal infrastructure support. We develop new techniques to cope with this dynamic, heterogeneous, and chaotic environment. In particular, we attempt to mask the unpredictable behavior by emulating a static *virtual* infrastructure that mobile nodes can interact with. The static virtual infrastructure allows for simpler algorithms — including many previously developed for fixed networks.

Virtual Stationary Automata programming layer. The static infrastructure consists of fixed, timed virtual machines, called *Virtual Stationary Automata* (VSAs), that are tiled over the entire plane. We develop a programming layer (which might be implemented as middleware) in which mobile nodes can take advantage of the virtual infrastructure to coordinate their actions. Each VSA represents a predetermined geographic area and has broadcast capabilities similar to those of the mobile nodes, allowing nearby VSAs and mobile nodes to communicate with one another.

Many practical algorithms depend significantly on timing, and it seems reasonable to assume that mobile nodes have access to a synchronized clock. In the VSA programming layer, the virtual automata, too, have access to a *virtual* clock. Moreover, the VSA programming layer guarantees that the virtual clock never drifts too far from the real clock. This requirement introduces significant difficulty in implementing the virtual infrastructure.

The virtual infrastructure that we propose differs in key ways from prior attempts to design abstractions for mobile ad-hoc networks. The GeoQuorums algorithm [9] proposes storing data at fixed locations; however it only supports atomic objects, rather than general automata. Another earlier attempt at defining a virtual infrastructure, the "Virtual Mobile Node Abstraction" proposed in [8], supports general automata; however the automata do not have access to a virtual clock. Moreover, the state machine replication algorithm described there cannot support a virtual clock. It is of interest to note that these abstractions could easily be implemented using the virtual infrastructure we describe here.

Emulating the virtual infrastructure. The VSA layer is emulated by the real mobile nodes in the network. In particular, a VSA is emulated by a bounded-size subset of the mobile nodes currently populating its geographic region: a mobile node that enters the geographic region of a VSA attempts to participate in the emulation of the region's VSA; a mobile node that leaves the geographic region ceases to emulate the VSA. If all the mobile nodes leave a VSA's region, then the VSA fails; if mobile nodes return, then the VSA restarts.

The state of the VSA is maintained in the memory of the participants, allowing them to perform actions on behalf of the virtual automaton. The implementation uses a "round-robin" approach in which participants take turns emulating the virtual automaton.

An important property of our implementation is that it is self-stabilizing. Self-stabilization [6], [7] is the ability to recover from an arbitrarily corrupt state. This property is important in long-lived, chaotic systems where certain events can result in unpredictable faults. For example, transient interference may disrupt the wireless communication, violating our assumptions about the

*Department of Computer Science, Ben-Gurion University, Beer-Sheva, 84105, Israel. Partially supported by IBM faculty award, NSF grant and the Israeli ministry of defense. Email: {dolev,lahiani}@cs.bgu.ac.il.

†MIT Computer Science and Artificial Intelligence Laboratory, The Stata Center 32-G670, Cambridge, MA 02139, USA. Supported by DARPA contract F33615-01-C-1896, NSF ITR contract CCR-0121277, and USAF,AFRL contract FA9550-04-1-0121. Email: {sethg, lynch, tnolte}@theory.csail.mit.edu.

broadcast medium. This might result in inconsistency and corruption in the emulation of the VSA. Our self-stabilizing implementation, however, can recover after corruptions to correctly emulate a VSA.

We present an algorithm that is a significant improvement over the prior attempts to emulate a virtual infrastructure described in [9], [8]. It is much more power efficient, limiting the number of participants to the minimum necessary to guarantee reliability. Moreover, the new algorithm reduces the number of messages broadcast and eliminates the duplicated messages that are possible in [8]. The current implementation also allows for faster emulation, introducing significantly less overhead into computation. Finally, the prior implementations are not self-stabilizing.

Applications. We present in this paper an overview of some applications that are significantly simplified by the VSA infrastructure. We consider both low-level services, such as routing and location management, as well as more sophisticated applications, such as pursuer identification and tracking. The key idea in all cases is to locate data and computation at virtual automata throughout the network, thus relying on the fixed, predictable infrastructure to simplify coordination. It is interesting to note that this infrastructure can be used to implement services that are oftentimes thought of as the lowest-level services in a network.

Our contributions. This paper contains three main contributions. First, we define a new VSA programming layer that supports timing dependent applications. Second, we present an energy efficient, self-stabilizing implementation of this virtual infrastructure. Finally, we discuss some applications that take advantage of the virtual infrastructure. We are currently working on understanding real-world implementation concerns.

Other prior work. There are a number of prior papers that take advantage of geography to facilitate the coordination of mobile nodes. For example, the GeoCast algorithms [19], [4], GOAFR [16], and algorithms for "routing on a curve" [18] route messages based on the location of the source and destination, using geography to delivery messages efficiently. A number of other papers [17], [12], [20] use geographic locations as a repository for data. These algorithms associate each piece of data with a region of the network and store the data at certain nodes in the region. This data can then be used for routing or other applications. All of these papers take a relatively ad hoc approach to using geography and location. In this paper we suggest a more general approach; all the algorithms presented in these papers would be simplified by using VSAs.

Organization. The rest of the paper is organized as follows. The system settings are described in Section II. We then define the virtual stationary automata (VSA) layer in Section III. Next we present a "round-robin" implementation of the virtual infrastructure in IV. We then describe some extensions and applications of VSAs in Section V.

II. Datatypes and Mobile Ad Hoc System Model

The system consists of a finite collection of mobile client processes moving in a closed region of the plane (see e.g., [9], [10]). In this section we formally describe the system, including: (1) datatypes used in the system description, (2) the model for the GPS automaton providing location and timing information to nodes, (3) the specification for a generic broadcast service, and (4) the model for the mobile clients deployed in the network.

A. Datatypes

Here we list the globally known constants. These define the regions (or tiles) of the netwotk, as well as the identifiers of the real mobile nodes:

- R, a fixed closed connected region of the two-dimensional plane. Our results should be extendable to larger dimensions given appropriate distance metrics in Section 3.
- U, a finite set of *region identifiers* for subregions of R.
- $nbrs$, a symmetric neighbor relation between elements of U.
- $m = |U|$.
- $region$, a mapping from U to connected subsets of R. We assume that $\{region(u) : u \in U\}$ forms a partition of R into *tiles*. In practice, one might want the regions that tile R to be regular polygons such as squares or hexagons.
- $regid$, a mapping from R to U, such that $regid(x, y)$ is equal to the unique $u \in U$ such that $(x, y) \in region(u)$.
- P, a finite set of *node identifiers*. We restrict P and U so that $P \cap U = \emptyset$.

B. GPS

The system is assumed to include a GPS automaton, providing location information to real mobile nodes. The GPS automaton is described formally in Figure 1. It is a timed I/O automaton (TIOA [15]) with access to a real time clock and that keeps the current location of all mobile nodes.

The main pieces of data kept by GPS are:

- $now \in \mathbb{R}$, a clock variable, representing real time.

Signature:
Output GPSupdate$(u)_p$, $u \in U, p \in P$
Internal updatefin

State:
analog *now* $\in \mathbb{R}$, current real time
sample $\in \mathbb{R}$, the next sample time, initially now
analog *loc*, an array of (x,y) coordinates in R, indexed by P
analog *vel*, an array of (x,y) velocities in R, indexed by P
updated, an array of Booleans, indexed by P, initially all `true`

Derived variables:
$maxdist(a, b) =$
if $a,b \in P$:
$|loc[a]\text{-}loc[b]|$
if $a,b \in U$:
$sup_{i \in region(a), j \in region(b)}\ |loc[i]\text{-}loc[j]|$
if $a \in U, b \in P$:
$sup_{i \in region(a)}\ |loc[i]\text{-}loc[b]|$
if $a \in P, b \in U$:
$sup_{i \in region(b)}\ |loc[i]\text{-}loc[a]|$

Actions:
Output GPSupdate$(u)_p$
Precondition:
$now = sample$
$updated[p] =$ **false**
$u = regid(loc[p])$
Effect:
$updated[p] \leftarrow$ **true**

Internal updatefin
Precondition:
$\forall p \in P.\ updated[p] =$ **true**
Effect:
$\forall p \in P.\ updated[p] \leftarrow$ **false**
$sample \leftarrow now + \epsilon_{sample}$

Trajectories:
satisfies
$\mathbf{d}(now) = 1$
$\forall p \in P.$
$|vel[p].x,\ vel[p].y| \le v_{max}$
$\frac{d(loc[p].x)}{dt} = vel[p].x$
$\frac{d(loc[p].y)}{dt} = vel[p].y$
constant *updated*, *sample*
stops when
$now = sample$

Fig. 1. GPS automaton

- loc, the continuously changing array of coordinates in R, indexed by node id.

We restrict the rate of change in locations, and hence the speed of the mobile nodes, to be at most v_{max}. To facilitate the task of updating processes with their region locations, we use two additional pieces of state:

- $sample \in \mathbb{R}$, the next sample time.
- $updated$, an array of Booleans indexed by process id, keeping track of whether an update for a particular process has occurred in this sample period.

We also define a derived variable based on the GPS state that is used throughout the paper:

- $maxdist$, an overloaded function from $(P \cup U) \times (P \cup U)$ to $\mathbb{R}$ that gives the distance between two nodes, the supremum distance between points in two regions, or the supremum distance between a node and a point in a region.

The GPS automaton has no input actions and performs only one kind of output and internal action:

- **Output** $\texttt{GPSupdate}(u)_p, u \in U, p \in P$: GPS is responsible for alerting each mobile node p of the identifier of the region u where p is currently residing. When a sample period deadline arrives ($sample = now$), the automaton performs a $\texttt{GPSupdate}(u)_p$ where $u = regid(loc[p])$, for each $p \in P$. It then updates $updated[p]$ to true, indicating the update has been performed.
- **Internal** `updatefin`: When all updates have occurred, the `updatefin` action changes $updated[p]$ to false for all $p \in P$ and sets the time for the next loc sampling to be ϵ_{sample} from now.

C. Broadcast service specification

Communication in the system is in the form of a local broadcast service. Here we provide a generic broadcast specification that is instantiated in the paper for various participants. We call the generic service bcast, parameterized by:

- r, the broadcast radius.
- d, the message delay.
- I, the set of port identifiers.

A service with these parameters is then called $\texttt{bcast}[r, d, I]$. It includes one piece of state:

- msg, an array of messages indexed by I.

The service provides one output and one input action:

- **Input** $\texttt{bcast}(m)_i$: The service allows for a port $i \in I$ to broadcast a message through $\texttt{bcast}(m)_i$. For each j in a set of identifiers $I' \subseteq I$, the $\texttt{bcast}(m)_i$ action puts a copy of m into a buffer $msg[j]$.
- **Output** $\texttt{brcv}(m)_i$: A message m in $msg[j]$ is delivered at port j through a $\texttt{brcv}(m)_j$ action, resulting in removal of m from $msg[j]$.

The service guarantees *reliable delivery* and *integrity*:

- Reliable delivery guarantees that if a port i transmits a message, then every port j such that

$maxdist(i, j) \leq r$ during the entire time interval starting from transmission and ending d time later receives the message within d time of transmission.
- Integrity guarantees that for any $\texttt{brcv}(m)_i$ that occurs, a $\texttt{bcast}(m)_j$ previously occurred, for some $j \in I$.

We also require that the service guarantees that for any messages $m \neq m'$, if a port broadcasts m and later broadcasts m', any port that receives both messages receives m before m'. This guarantee is assumed for convenience; it is possible to guarantee this property by supplementing messages with additional information to allow recipients to reorder broadcasts from the same sender. However, this is not the focus of the paper, so we simply assume we are guaranteed this property.

In practice, a broadcast service would have bounded message buffers. Bounded buffers are also necessary to guarantee self-stabilization in the face of corruption errors. Here we assume that in the event of buffer overflow, overflow messages are lost. Buffer sizes are oftentimes chosen by considering the maximum node density of the network and maximum frequency of message broadcasts. Here we assume the buffers are sufficiently large that overflows do not occur in normal operation.

The model can be extended to incorporate collisions. A brief discussion of the impact of collisions on our work can be found in Section V-A.

D. *Client nodes and P-bcast*

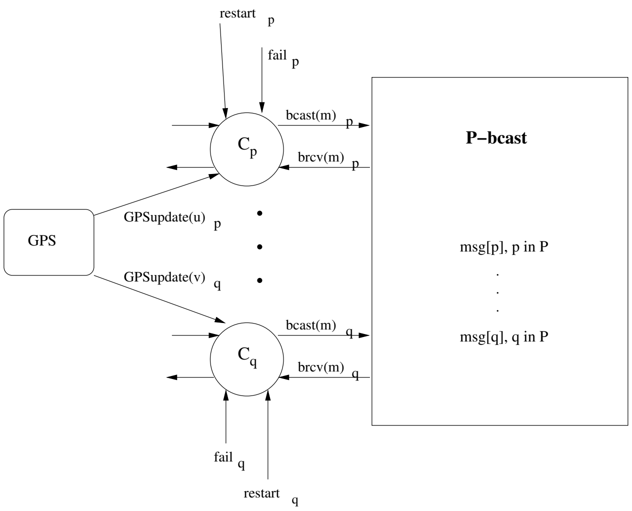

Fig. 2. P-bcast and client node interface. Client nodes can receive GPS updates and communicate together through the P-bcast service.

For each $p \in P$, we assume a timed I/O automaton C_p (Figure 2) with access to the following local variable:

- $now \in \mathbb{R}$, a clock variable, representing real time and synchronized across clients. For simplicity's sake, we treat now as a local variable that progresses exactly like real time. This now variable could, alternatively, be a variable frequently updated by a GPS automaton.

Client also have access to a local broadcast service that is defined as an instantiation of the generic bcast service described in the last section, called P-bcast and parameterized by the following:

- r_{real}, the broadcast radius.
- d, the message delay.
- P, the node ids, meaning there is one communication port per process.

A client C_p is assumed to have at least the following external interface, allowing the client to broadcast and receive messages and receive GPS region updates. The interface also allows the possibility that nodes may crash-stop and later restart:

- **Output** $\texttt{bcast}(m)_p$: A node p may broadcast a message through $\texttt{bcast}(m)$.
- **Input** $\texttt{brcv}(m)_p$: A node p receives messages through $\texttt{brcv}(m)$.
- **Input** $\texttt{GPSupdate}(u)_p$: GPS updates for a node p occur through the $\texttt{GPSupdate}(u)_p$ action, indicating that p is currently located in region u.
- **Input** $\texttt{fail}_p$: A node p can be halted by a $\texttt{fail}_p$ input and then performs no local steps unless it later recovers through a $\texttt{restart}_p$ input.
- **Input** $\texttt{restart}_p$: The $\texttt{restart}_p$ action restarts the node automaton from an initial state.

In addition we assume that there may exist arbitrary input and output actions with the external environment and there may be other pieces of local state used by an algorithm running at the node.

For convenience we assume local steps take no time. Also, for simplicity of presentation, we assume for now that the nodes do not suffer from corruption failures. When a node suffers from a corruption failure, the node suffers from nondeterministic changes to its program state. We discuss the case where corruption faults may occur in Section IV-D.

III. VIRTUAL STATIONARY AUTOMATA PROGRAMMING LAYER

The *Virtual Stationary Automata* programming abstraction includes both the real mobile nodes discussed in the last section and virtual stationary automata (VSAs) the real nodes emulate, as well as a local broadcast service, V-bcast, between them (see Figure 3). This abstraction allows users to write programs for stationary

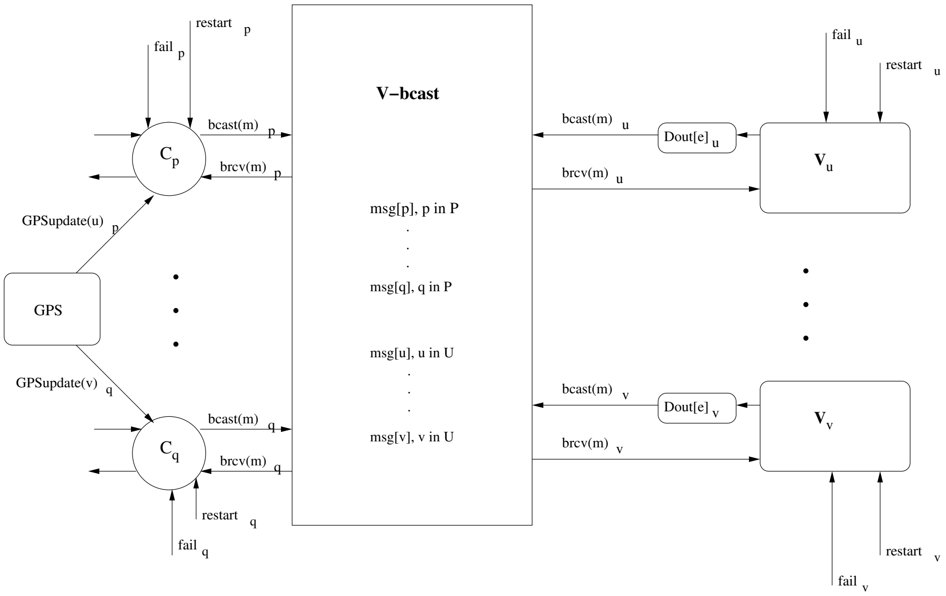

Fig. 3. Virtual Stationary Automata abstraction. VSAs and clients communicate using the V-bcast service. VSA bcasts may be delayed in Dout buffers.

regions of the network as though broadcast-equipped virtual machines exist in those regions. In this section we define what we can support in this layer, given that the VSAs and the V-bcast service must be implemented by the underlying real mobile nodes.

Here we describe the properties of VSAs we can support. We then describe the V-bcast service. The V-bcast service is similar to the mobile nodes' P-bcast service except that: (1) it allows communication between neighboring VSAs and nearby mobile nodes, and (2) the broadcast radius supported is slightly smaller than that of P-bcast, for reasons we will explain. Finally, a VSA is emulated by real mobile nodes that coordinate their emulation and may fail. This emulation by real nodes can introduce delays in the emulation of the VSA which we describe with the concept of *delay augmentation*.

A. Virtual Stationary Automata

A VSA is an abstraction describing a virtual machine that may be emulated by the underlying real mobile nodes residing in particular regions in the network. The VSA is permitted to use timing. We formally describe such a timed machine V_u as a TIOA whose program can be referred to as a tuple of its action signature, valid states, its start state, a discrete transition function, and the set of valid trajectories of the machine: $\langle sig, states, start, \delta, \tau\rangle$. Trajectories [15] describe the evolution of the state over intervals of time.

To guarantee that we can emulate the machine we must guarantee that inputs and outputs are such that they can be emulated by the real mobile nodes. Hence, the automaton's external interface is restricted to include only failure/restart inputs and the ability to broadcast and receive messages. In other words, we restrict this virtual automaton V_u to have only four external actions:

- **Input** $\texttt{fail}_u$: A VSA can be crashed by a $\texttt{fail}_u$ input, making no local steps unless it later recovers.
- **Input** $\texttt{restart}_u$: A failed VSA makes no local steps unless it later recovers through a $\texttt{restart}_u$ input, resulting in a reset of $vstate$ to a state in $start$.
- **Input** $\texttt{brcv}(m)_u$: The VSA at region u receives a broadcast message m from the V-bcast service.
- **Output** $\texttt{bcast}(m)_u$: The VSA broadcasts a message m through the V-bcast service.

The current state of all variables of V_u can be referred to collectively and is assumed to include a variable corresponding to real time:

- $vstate \in states_u$, the current state of V_u.
- $vstate.now \in \mathbb{R}$, the clock time of V_u.

While we do not explicitly do so in this section for presentation reasons, the VSA programming layer has been extended to incorporate corruption failures. This is modeled using an additional input action called $\texttt{corrupt}_u$, resulting in a nondeterministic change to any portion of $vstate$ except $vstate.now$. A $\texttt{corrupt}$ action at this layer is restricted to only occur if a $\texttt{corrupt}$ action occurs in the mobile node layer. If the model is extended to incorporate corruption failures, users of the model must be careful to program V_u with the possibility of corruption in mind, meaning programs for V_u must be self-stabilizing.

B. V-bcast service

The V-bcast service is a local broadcast service similar to that of the mobile nodes' P-bcast service and implemented using the real mobile nodes and the P-bcast service. It allows communication between neighboring VSAs and between VSAs and nearby nodes. It supports a slightly smaller broadcast radius than that of P-bcast.

We again define the V-bcast service as an instantiation of the bcast service. In this case it is instantiated with:

- r_{virt}, the broadcast radius.
- d, the message delay, equal to that of P-bcast.
- $P \cup U$, meaning there is a communication port for every process and virtual automaton.

The V-bcast service provides the same guarantees as the generic bcast specification described in the prior section. This service allows a VSA for region u and a real mobile node p to communicate as long as the node is at most r_{virt} distance from any point in region u and a VSA to communicate with another VSA as long as the maximum distance between points in either VSA is at most r_{virt}.

In order to guarantee that two neighboring VSAs may communicate we require that the tiling described by $region$ and the region neighbor relation $nbrs$ satisfies the restriction that for any neighboring regions $u, v \in U$, the supremum distance between a point in u and a point in v is at most r_{virt}.

The implementation of the V-bcast service using the mobile clients' P-bcast service also introduces the requirement that $r_{real} \geq r_{virt} + 2\epsilon_{sample} \cdot v_{max}$. This guarantees that two nodes that are unknowingly emulating VSAs for regions they have just left (because they have not yet received `GPSupdates` to change regions) can still receive messages transmitted to each other.

Messages intended for a node p are stored in buffer $msg[p]$ until delivery and messages intended for a VSA u are stored in $msg[u]$.

C. Delay augmentation

A VSA V_u is an abstraction that is emulated by underlying real mobile nodes. While the resulting emulation of V_u would ideally look identical to a legitimate execution of V_u, the abstraction must reflect the possibility that, due to delays resulting from message delay or real node failure, the emulation of V_u might be slightly behind real time and appear to be delayed in performing output actions of V_u by up to some time that we'll call e. The emulation of V_u is then called a *delay-augmented TIOA*, an augmentation of V_u with timing perturbations composed with V_u's output interface. These timing perturbations of up to e time are represented with a buffer Dout$[e]_u$, composed with V_u's `bcast` output. The buffer is a message multiset that delays delivery of messages by some nondeterminstic time $[0, e]$. The program actions of V_u must be written taking into account the emulation parameter e, just as it presumably takes into account the message delay factor d.

IV. Round-robin implementation of a VSA

We describe the implementation of an abstract VSA by mobile clients in its region. We then give several proof sketches and discuss the self-stabilizing extension of the implementation.

A. Implementation description

Here we describe, at a high leve,l a fault-tolerant implementation of a VSA emulator. We begin by describing a single emulator solution. We then extend the solution to be fault-tolerant by using multiple emulators and describe a *round-robin* mechanism to manage the multiple emulators.

Single emulator solution. In the simplest version of virtual machine emulation, there is one designated mobile node called a *leader* that emulates the VSA. The leader has sole responsibility for the emulation, performing all actions of the VSA based on its locally stored version of the VSA state, identical to the state of the abstract VSA, and messages it has received.

Multiple emulator extension. For fault-tolerance and load balancing reasons, it is necessary to have more than one process emulating a VSA. In the more general multiple emulator approach a VSA for a region u is emulated by up to k (a constant) mobile nodes located in region u called *guards*. At any time, there is at most one guard that is designated leader that emulates the VSA. As before, the leader emulates actions of the VSA based on its locally stored version of the VSA state and messages it has received. Occasionally the leader hands off responsibility for emulating the VSA to another guard by sending a special hand-off message, containing a copy of the leader's current emulated VSA state. When a guard receives this message it updates its local copy of the VSA state to be consistent.

This hand-off introduces several complications in the VSA emulation, resulting both from message delivery delays and failures of leaders that are emulating the VSA. The first is that a message m could be `bcast` to the VSA but not received by the leader before it `bcasts` a hand-off message; if non-leaders do not save such messages they would hear m, discard it, receive the hand-off message, and (if now a leader) continue to emulate the VSA as though m was never received. To prevent the emulation from failing to eventually process messages, all guards save received messages locally for use when it is their turn to emulate the VSA. To ensure that multiple-processing of messages at the VSA does not result, the emulators need to know which of the saved messages have already been processed by the leader. To answer this, the leader sends more than its emulated VSA state with a hand-off message — it also sends a queue of the messages it processed for the VSA while leader. When a guard receives the hand-off message it updates its local VSA state as before and then deletes those messages already processed by the VSA from its local queue of messages to be processed.

Message delays complicate matters for another reason as well. A leader could, for example, receive a message m, process it, and send a hand-off message. The next leader could receive the leader's message, update its local state, and *then* receive the message m. It would then re-process m, which is undesirable. This is easily solved by having all messages timestamped with the time they were sent and all guards wait for the full d message delay time after a message's timestamp before handling it for any reason. Any message received or handled is guaranteed to have been seen by other processes as well. This proves to be useful later in several places.

Due to message delays that occur during hand-off and additional delays that can occur because of failures of leaders, the emulation of the VSA may be behind in real time by a considerable amount. Intuitively, the VSA emulation runs on a virtual clock that is stopped whenever a hand-off message is in transit and whenever no leader is currently emulating the VSA. In order to guarantee that the VSA emulation satisfies the specifications from Section 3, the virtual clock must be able to catch up to real time during periods when the VSA emulation is running (the specification bounds the amount of time the output trace of the VSA emulation may be behind that of the VSA being emulated by a parameter e). This is done by having the virtual clock advance faster than real time until both are equal, at which point they increase at the same rate. This is illustrated in Figure 4, where the progress of the virtual clock proceeds in fits and starts relative to real time, occasionally falling behind and then having to catch up. The magnitude of speed-up of the virtual clock is dependent on how long the leader hand-offs take and how often guards fail or leave the region.

A related problem occurs in the processing of messages. Since the virtual time may be behind real time, it is possible that there are messages that should eventually be processed but that were sent at a real time after the

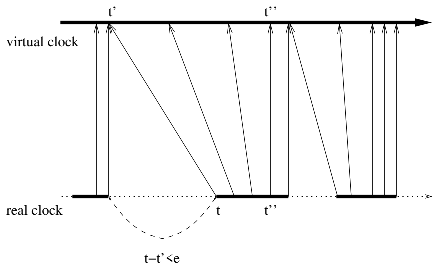

Fig. 4. Relationship between virtual and real time. Virtual clocks that are behind real clocks run faster until they catch up.

virtual time. Delivery of future messages from the perspective of the emulated VSA is undesirable; it violates the requirement that the VSA emulation produces an execution that looks like one of the abstract VSA with additional message delays. This problem is solved by simply waiting to process a message until the virtual clock passes the message's timestamp.

Another complication results when a leader `bcasts` messages on the VSA's behalf but fails before sending a hand-off message. In this case, the next leader emulating the VSA has an out-of-date version of both the emulated VSA state and the queue of messages already "received" by the VSA. Based on this out-of-date information, the new leader may re-perform actions already performed by a prior leader. As a result, the external trace, and `bcasts` in particular, might not be consistent with a trace of the abstract VSA. To handle this, when a leader performs a `bcast` action for the VSA it attaches its version of the post-`bcast` VSA state and processed messages to the message. If the leader fails to transmit a hand-off, the next leader will pick up emulation at the state attached to the last output, guaranteeing consistent traces.

Round-robin emulator management. The multiple emulator approach relies on a fault-tolerant algorithm for managing leadership. In the *round-robin* approach this is done by defining globally known synchronized *timeslices* and maintaining a k-bounded rotating $guards$ vector of process id/timestamp pairs, defining revolving responsibility for VSA emulation. The timestamp is the time when the guard requested to join the vector and is explained later. Whichever guard's pair is currently at the head of the rotating vector is designated the leader.

Each timeslice is of a predetermined length t_{slice} such that $t_{slice} \leq e/k$ and $t_{slice} > d$. A *round length* is the amount of time it takes for k timeslices to pass.

For consistency reasons, as before, any `bcasts` for the VSA performed by the leader have attached to them the leader's latest version of the emulated VSA state and the messages processed by the VSA during the leader's emulation period. Now, to keep leadership views consistent, we also transmit the leader's $guards$ vector. The tuple of the three pieces of information is called the *emulation state*. When a guard receives the emulation state, in addition to prior changes to its local state, it updates its $guards$ vector to match that of the leader. It uses the new $guards$ vector to clean up too old or unsuccessful requests to join the $guards$ vector (described later).

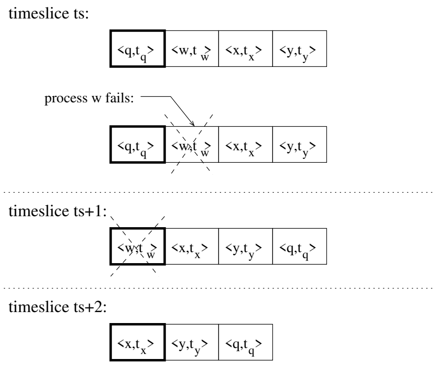

Fig. 5. Handling failures in the guards vector. Process w fails in timeslice ts. In timeslice ts+1, after the vector rotates, w is supposed to be leader. Other guards do not receive a hand-off from w and remove w from the vector at the beginning of timeslice ts+2.

At the end of a timeslice, the leader broadcasts a hand-off message. It then becomes a regular guard. All guards should receive this message by d time into the next timeslice, if it was sent. If it was not, then all guards will remove the previous leader's entry from their local versions of the $guards$ vector (Figure 5).

All guards then rotate the vector once. The new head of the $guards$ vector becomes leader for the timeslice and starts emulating the VSA based on its local version of the emulation state. In order to make up the time that is lost between the last sending of the emulation state by a leader and its own pick-up of the emulation, the new leader emulates the VSA using a sped-up virtual clock as described before.

The magnitude of the speed-up is determined as follows: Assume that we are considering a VSA emulation where at least one leader completes his timeslice in each round. With this assumption, the furthest that the virtual clock could be behind when a leader starts emulating the VSA is $(k-1) \cdot t_{slice} + d$, since at worst $k-1$ leaders in a row could have failed without sending any emulation state, followed by d time for the one alive leader to start emulating the VSA again. To ensure that by the end of the timeslice $t_{slice} - d$ later the virtual clock has caught up to real time, the virtual clock must emulate a total of $k \cdot t_{slice}$ time in the time that the leader is emulating the VSA, namely $t_{slice} - d$ time. Together, this gives that the leader must advance the virtual clock at a speed of at least $s > \frac{k \cdot t_{slice}}{t_{slice} - d}$ times the rate of real time.

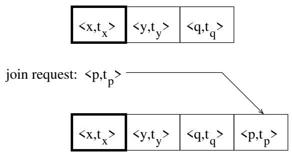

Fig. 6. Adding a join request to the guards vector. Process p sends a join request at time t_p. If the vector is not too large, the leader processes the request, adding $\langle$p, $t_p\rangle$ to the end of the guards vector.

A process that enters a region attempts to become a guard by broadcasting a special `join` message and then collecting messages just as guards would. All guard processes save the process id and time the message was sent in a local queue of join requests. A leader processes a request from its local queue by adding the process id and timestamp pair to the end of its local $guards$ vector, if there is room (Figure 6). A process p that started trying to join at time t examines any messages broadcast by the leader for the attached emulation state's $guards$ vector to determine if its pair $\langle p, t\rangle$ has been added. If so, it changes its status to be `guard`. If not and enough time has passed since its request that the leader would have added it if there was room, p just tries to re-join.

Notice that when joining, process p only deems itself successful if it sees its $\langle p, t\rangle$ pair in the $guards$ vector, rather than any pair $\langle p, t', \rangle$ where $t \neq t'$. This deals with the case where p might be a guard in region u, leave the region, and then try to rejoin the region before it is removed from the $guards$ vector. If p were to immediately start emulating the VSA at this point, it would run the risk of not having received and queued all messages for the VSA that it should have. Instead p waits to see that its newest join request is reflected in the $guards$ vector before becoming a guard. This is safe since its join request is not seen by it in a $guards$ vector until at least $2d$ time after p sent it (due to mandatory waiting before delivering messages), guaranteeing it has collected all the broadcast messages that are not summarized in the emulation state.

If a process tries to join but a round goes by without it hearing any broadcasts by a VSA emulator, it concludes there are no emulators for the VSA. In this case, it broadcasts a `restart` message and collects other restart messages that are broadcast. The senders of these messages are sorted by id in the $guards$ vector and the one with the lowest id becomes leader in the next round.

B. Detailed code description

The emulators for the VSA (VSAEs) run on individual mobile nodes. Formally, there exists one emulator automaton $\text{VSAE}_{u,p}$ for each pair $(u, p) \in U \times P$. This automaton handles mobile node p's portion of the emulation of V_u. Here we describe in detail the actions described in Figure 8, the locally checked and corrected actions, and the trajectories described in Figure 9.

Discrete action descriptions. We begin by describing the code for $\text{VSAE}_{u,p}$ in Figure 8.

- `GPSupdate`(v), Line 1: This input indicates process p is in region u. Process p changes its reg to the region v. If the region is different from p's previous region, p changes its $status$ to `startjoin`, which in turn enables the `bcast`($\langle\langle$join$, u\rangle, p, now\rangle$) action.
- `brcv`(msg), Line 7: Messages sent to and from the VSA are of a special form. In particular, they are three-place tuples, including the message one wants to send, the source (whether it be a VSA or a client), and the timestamp of the message. For convenience, we refer to these portions of the message msg as $msg.m, msg.src$, and $msg.ts$ respectively. We will assume in the implementation of the VSAs that any messages received through the `brcv` action are of this special form. We note that any messages not of this form can simply be "filtered out."
  When a process p receives such a message msg from a client or from a neighboring VSA through the `brcv`(msg) action, it places the message into the $holdq$ queue.
- `bcast`($\langle\langle$join$, u\rangle, p, now\rangle$), Line 12: This broadcasts a join request by p for the VSA at u. This changes p's status to `trying`, sets its $joinreqts$ timestamp (used to keep track of when it asked to join the $guards$ vector), sets the start time for the next global timeslice, sets $round$ (a deadline to determine that no guards are emulating the VSA), and initializes its $guard$ vector and its $simq, holdq, guards$, and $joinreqs$ queues.
- `bcast`($\langle\langle$restart$, u\rangle, p, now\rangle$), Line 23: If the VSA has failed, the joining node (p) will receive no broadcasts from guards for k time slices. After the $round$ deadline is reached, the node broadcasts a `restart` message. This results in a reset of p's $joinreqts$ to the current time and an emptying of the $guards$ vector, in preparation for starting a new one.
- `delayrcv`(msg), Line 32: When d time has passed from the timestamp of msg, the `delayrcv`(msg) action removes the message from $holdq$ and handles it depending on the kind of message.
  If the message is a restart message and p has a status of `trying` and a $round$ deadline that has passed, p places the sender's id with the message timestamp into its $guards$ vector, sorted in ascending order by id. If p is either not `trying` or its $round$ deadline has not passed, it tries again to join.
  The `delayrcv` action processes a join request message by adding the join request id and timestamp pair to its local $joinreqs$ queue.
  For any process, when `delayrcv`$_i$ handles a message in $holdq$ that is not a `join`, `restart`, or `end` message, it puts it into the local $simq$, which acts as a virtual message buffer for the VSA.
  If the message's source $msg.src$ is the VSA for p's region, the emulation state attached to the message is used to update p's emulation state. If the received $guards$ vector indicates a different leader than the one p currently has (and the $guards$ vector isn't empty, which only happens for `trying` processes that have not processed any emulation state message yet), it re-joins; something went wrong someplace for this to happen. If p is not the leader, it copies the $vstate$ and $guards$ vector indicated in the emulation state (the leader does not copy over his up-to-date state of emulation with the outdated state it may have sent d time ago, since the emulation state should have progressed since that time). For any status, p updates its $simq$ by cleaning out those messages already processed by the leader and those messages that are simply too old relative to the time when the state was sent. Similarly, it cleans out its list of outstanding join requests by removing those join requests from its local $joinreqs$ that are already reflected in the $guards$ vector, as well as those requests that are old enough that they would have been seen by the leader.
  If the $guard$ vector incorporates p's join request and p is still `trying`, then its status becomes `guard` and $leadup$ gets initialized to **false**. If, however, the node's request is not reflected and the message's timestamp is more than d time after its join request then the node restarts its join.
  If the special `end` message is received, the $leadup$ variable is updated to indicate that the leader sent out an end-of-timeslice message.
- `tsBegin`, Line 68: In action `tsBegin`, d time after the beginning of a timeslice, all guards perform some $guards$ vector upkeep. If $leadup$ is **false**, indicating the leader failed in the last timeslice, the head of the $guards$ queue is dropped. Otherwise, the vector is rotated once.
  If p is still `trying`, it's id and $joinreqts$ are in the $guards$ vector, and the deadline for hearing from a guard has passed, then the VSA emulation has been restarted and p is a guard. As a result p changes its status to `guard`, starts the VSA again in an initial state, and initializes the list of outstanding

Bcast Messages:
$M' = M \times (P \cup U) \times \mathbb{R}$, where M may be arbitrary.
For convenience, we view $msg \in M'$ as a record:
$msg = \langle m, src, ts \rangle$.
We allow the use of $msg.m.first$ (or $second$) to access the first or second field of m in the event m is a tuple.

Signature:
Input GPSupdate$(v)_p$, $v \in U$
Input brcv$(msg)_p$, $msg \in M'$
Output bcast$(msg)_p$, $msg \in M'$
Internal delayrcv$(msg)_{u,p}$, $msg \in M'$
Internal tsBegin$_{u,p}$
Internal joinhandle$(\langle q, t \rangle)_{u,p}$, $q \in P$, $t \in \mathbb{R}$
Internal VSArcv$(m)_{u,p}$, $m \in M'$
Internal VSAint$(act)_{u,p}$, $act \in$ **internal** actions of sig_u

State:
vstate $\in states_u$, the state of V_u
analog *now* $\in \mathbb{R}$, the current real time
reg $\in U$, p's region location, initially $init(p)$
status $\in$ {guard, leader, trying, startjoin}, initially startjoin
timeslice $\in \mathbb{R}$, the next timeslice
round $\in \mathbb{R}$, a deadline for a new round
holdq, a queue of messages in M' without repetition
simq, a queue of messages in M' without repetition
procedq, a queue of messages in M'
guards, a vector of pairs of ids in P and times (of form $\langle id, ts \rangle$), of size at most k
joinreqs, a vector of pairs of ids in P and times (of form $\langle id, ts \rangle$)
joinreqts, the time of the last join request
leadup, a Boolean

Fig. 7. VSAE$_{u,p}$ emulating V$_u$ running $\langle sig_u, states_u, start_u, \delta_u, \tau_u \rangle$: Signature and State

join requests and the queue of messages intended for the VSA.
If p is the head of the vector and has status of `guard`, it changes its $status$ to `leader`.
The $leadup$ variable is reset to **false**, the $procedq$ is cleared for the timeslice, and the time for the next timeslice is stored.

- `joinhandle`$(\langle q, t \rangle)$, Line 87: When the leader processes a join request $\langle q, t \rangle$ in its local $joinreqs$ queue, it cleans out older entries for the same process q. If the vector of guards is smaller than k the leader adds q's id and join request time to the end of the vector. If the vector is full, the join request is simply removed.
- `VSArcv`(m), Line 98: The leader emulates receipt of VSA messages by performing `VSArcv`(msg) actions on messages sent no later than time $vstate.now$ in its local $simq$. The action removes msg from the $simq$ and emulates the receipt of $msg.m$ at the VSA. The resulting change of the state of the VSA is stored in $vstate$. The message msg is then added to $procedq$, the queue of messages "received" by the VSA in this timeslice.
- `VSAint`(act), Line 108: A valid internal action act of the VSA is emulated with the `VSAint`(act) action at the leader. The action results in a change of $vstate$ to the resulting state of the VSA.
- `bcast`$(\langle\langle m, \langle, vstate', procedq, guards \rangle\rangle, u, now \rangle)$, Line 116: A broadcast by the VSA of a message m is emulated through a `bcast`$(\langle\langle m, \text{emulation state} \rangle, u, now \rangle)$ action at a leader. We attach the post-broadcast VSA emulation state to the message being sent.
- `bcast`$(\langle\langle \text{end}, \langle vstate, procedq, guards \rangle\rangle, u, now \rangle)$, Line 124: The leader performs the emulation until the end of his timeslice and no outstanding requests or messages exist, at which point it again broadcasts the emulation state through a `bcast`$(\langle end, \text{emulation state}, u, now \rangle)$ action and changes to being just a `guard`.

Trajectories. The trajectory in Figure 9 describes the development of the variables in the implementation outside what is described through the discrete actions.

Of particular interest are lines 5 through 8, which dictate that, if a leader, the virtual clock that is behind real time in the emulation runs $s > \frac{k \cdot t_{slice}}{t_{slice} - d}$ times faster than the real clock, guaranteeing that the maximum break between the broadcasting of emulation state between two leaders in an alive VSA can be overcome in one leader's timeslice. Once the fast virtual clock catches up to real time, the virtual clock progresses as real time until the end of the timeslice, where the leader gives up leadership.

In line 11, we relate the emulated machine's trajectories τ_u to the emulator's trajectories τ. This line states that if we examine the current trajectories of the emulator, $vstate$ is the same as the trajectory that would have been observed at the emulated machine at time $vstate.now$.

The stopping conditions described in the second column are a means by which to force discrete actions in Figure 8 to occur when they are enabled.

Client bcast and brcv. Client broadcasts and receives are implemented using the P-bcast service. To distinguish messages as V-bcast messages, we use messages of the special form used above. In particular, a client at node p implements a `bcast`$(m)_p$ in the V-bcast service by performing a `bcast`$(\langle m, p, now \rangle)_p$ in the P-bcast service. The same client implements a `brcv`$(m)_p$ in the V-bcast service if it performs a `brcv`$(\langle m, u, t \rangle)_p$ in the P-bcast service where $u = reg_p$ and $t \in \mathbb{R}$.

```
Input GPSupdate(v)_p
Effect:
  if reg ≠ v then
    status ← startjoin
  reg ← v

Input brcv(msg)_p
Effect:
  if (msg.src ∈ P or msg.src ∈ nbrs(u)) then
    holdq ← holdq ∪ {msg}

Output bcast(⟨⟨join, u⟩, p, now⟩)_p
Precondition:
  reg = u
  status = startjoin
Effect:
  status ← trying
  joinreqts ← now
  timeslice ← nextTS(now)
  round ← timeslice + k ·t_slice + d
  simq, holdq, guards, joinreqs, procedq ← ∅

Output bcast(⟨⟨restart, u⟩, p, now⟩)_p
Precondition:
  reg = u
  status = trying
  now = round
Effect:
  joinreqts ← now
  guards ← ∅

Internal delayrcv(msg)_{u,p}
Precondition:
  msg ∈ holdq
  msg.ts = now-d
Effect:
  holdq ← holdq / {msg}
  if msg.m = ⟨restart, u⟩ then
    if (status = trying and round < now) then
      insertsort(guards, ⟨msg.src, msg.ts⟩)
    else status ← startjoin
  else if msg.m = ⟨join, u⟩ then
    joinreqs ← joinreqs ∪ {⟨msg.src, msg.ts⟩}
  else if (msg.m).first ≠ end then
    simq ← simq ∪ {m}
  if msg.src = u then
    let ⟨vstate′, procedq′, guards′⟩ = (msg.m).second in
      if (head(guards) ≠ head(guards′) and guards ≠ ∅) then
        status ← startjoin
      if status ≠ leader then
        vstate ← vstate′
        guards ← guards′
      simq ← simq / procedq′
      simq ← simq / {ms: (ms.ts < msg.ts-d  and
                              ms.ts ≤ vstate.now)}
      joinreqs ← joinreqs / guards
      joinreqs ← joinreqs / {⟨q, t⟩ : t < msg.ts-d}
      if ⟨i, joinreqts⟩ ∈ guards then
        if status = trying then
          status ← guard
          leadup ← false
      else if joinreqts < msg.ts-d then
        status ← startjoin
      if (msg.m).first = end then
        leadup ← true

Internal tsBegin_{u,p}
Precondition:
  now = timeslice + d
Effect:
  if status = guard then
    if leadup = false then
      guards ← remove(guards, head(guards))
    else guards ← rotate(guards)
  if (status = trying and ⟨p, joinreqts⟩ ∈ guards and round < now)
                                                              then
    vstate ← start_u
    simq, joinreqs ← ∅
    status ← guard
  if (status = guard and ⟨p, joinreqts⟩ = head(guards)) then
    status ← leader
  leadup ← false
  procedq ← ∅
  timeslice ← nextTS(now)

Internal joinhandle(⟨q, t⟩)_{u,p}
Precondition:
  status = leader
  ⟨q, t⟩ ∈ joinreqs
Effect:
  while ∃ t′ ∈ ℝ: (t > t′ and ⟨q, t′⟩ ∈ guards)
    guards ← guards / {⟨q, t′⟩}
  if (|guards| < k and ∄ t′ ∈ ℝ: ⟨q, t′⟩ ∈ guards) then
    append(guards, ⟨q, t⟩)
  joinreqs ← joinreqs / {⟨q, t⟩}

Internal VSArcv(m)_{u,p}
Precondition:
  status = leader
  m ∈ simq
  m.ts ≤ vstate.now
Effect:
  vstate ← δ_u(vstate, brcv(msg.m))
  simq ← simq / {m}
  procedq ← procedq ∪ {m}

Internal VSAint(act)_{u,p}
Precondition:
  reg = u
  status = leader
  δ_u(vstate, act) ≠ ⊥
Effect:
  vstate ← δ_u(vstate, act)

Output bcast(⟨⟨m, ⟨vstate′, procedq, guards⟩⟩, u, now⟩)_p
Precondition:
  reg = u
  status = leader
  δ_u(vstate, bcast(m)) = vstate′ ≠ ⊥
Effect:
  vstate ← vstate′

Output bcast(⟨⟨end, ⟨vstate, procedq, guards⟩⟩, u, now⟩)_p
Precondition:
  reg = u
  status = leader
  now = timeslice
  simq, joinreqs = ∅
Effect:
  status ← guard
```

Fig. 8. $\mathrm{VSAE}_{u,p}$ emulating V_u running $\langle sig_u, states_u, start_u, \delta_u, \tau_u\rangle$: Actions

satisfies
  d(*now*) $=1$
  constant *timeslice*, *round*, *status*, *reg*, *simq*, *holdq*,
    procedq, *guards*, *joinreqs*, *leadup*, *joinreqts*
  if *status* $=$ leader **then**
    if *vstate.now* $<$ *now* **then**
      d(*vstate.now*) $> \frac{k \cdot t_{slice}}{t_{slice}-d}$
    else *vstate.now* $=$ *now*
  else constant *vstate*
τ **also satisfies**
  τ(*now*).*vstate* $= \tau_u(\tau$(*now*).*vstate.now*)

stops when
  reg $=$ *u* **and**
  { $\exists$ *msg* $\in$ *holdq*: [*msg.ts* $=$ *now* -*d*] **or**
   now $=$ *timeslice* + *d* **or**
   status $=$ startjoin **or**
   (*status* $=$ leader **and** (*now* $=$ *timeslice* **or**
    joinreqs $\neq \emptyset$ **or** $\exists$ *m* $\in$ *simq*: *m.ts* $\geq$ *vstate.now*)) **or**
   (*status* $=$ trying **and** *now* $=$ *round* **and** *joinreqts*$<$ *now*) }

Fig. 9. VSAE$_{u,p}$ emulating V$_u$ running $\langle sig_u, states_u, start_u, \delta_u, \tau_u \rangle$: Trajectories

C. *Proof sketches*

We sketch the proof that the emulator implementation is correct. First, we show that the implementation manages guards sensibly. We then demonstrate a forward simulation relation [15] between the implementation and the VSA abstraction described in Section III, implying the VSA emulator correctly implements the VSA abstraction.

For the rest of this section, consider one region u and its corresponding VSA V_u and an execution where each process p in region u starts with knowledge it is in u. For simplicity, we do not consider corruption faults here.

Guards management. The implementation guarantees certain properties of the $guards$ vector. We can show the following lemmas:

Lemma 4.1: At most one process is a leader and at most k are either a leader or guard.

Lemma 4.2: A process that is a guard or leader remains a guard or leader until it leaves the region or fails.

Lemma 4.3: A process that is a guard and remains alive and in the region for k timeslices will be a leader in at least one of those timeslices.

The next lemma guarantees that, subject to certain assumptions about mobile node movement and failure, some processes will become guards, which is necessary for an emulation to be of a non-failed VSA:

Lemma 4.4: If there are fewer than k guards and leaders and a set of processes P' that are trying to become guards that remain alive in the region for "long enough", then a nonempty subset of P' become guards.

Proof sketch: The proof has two main cases. The first is where no processes are guards or leaders: Consider the id-ordered subset of P' that remains alive through d into the $k+1^{st}$ next full timeslice. If any of the first k in the subset remains alive for another timeslice, then they become guards. This is through `restart` messages. The second is the easier case where there is a guard or leader that remains alive long enough to add join requests to the $guards$ vector. ■

Simulation relation. The next step is to show through use of a forward simulation relation and history variables that the emulation results in a correct implementation of the VSA abstraction, allowing applications built for the VSA abstraction to run on the VSA emulators.

We define the emulation of a VSA V_u as *failed* during an execution fragment if there is a state where there is no process that is a guard or leader. We now define the simulation relation on states where the emulation has not failed. It consists of several parts, relating state of emulators to the state of the abstract VSA and state of message buffers in the implementation to those of the abstract system.

If process p is not a leader and there is a message m from the VSA with attached emulation state containing $vstate'$ in P-bcast.$msg[p]$ or p's queue of messages it is waiting to deliver, then $vstate'$ from the latest such message is equivalent to $V_u.vstate$. If no such message exists and p is a guard, then p's own local value of the virtual state is equivalent. If there is only a leader and no guards, then the leader's local version of the virtual state is equivalent to $V_u.vstate$.

If m is a message in P-bcast.$msg[p]$, in p's queue of messages it is holding until old enough, or in p's queue of messages it is saving for the VSA such that it is not a processed message in the emulation state of some message in transit, and if m was sent no later than the time on the virtual clock as figured from the virtual state above, then it is also waiting in V-bcast.$msg[u]$.

Lastly, if m is a message in P-bcast.$msg[p]$ that was sent by V_u then the message is either in Dout$[e]_u$ (if the virtual clock as figured from above is behind the timestamp of the message) or else in V-bcast.$msg[p]$.

Using the simulation relation we can prove the main theorem by induction on implementation actions:

Theorem 4.5: The VSA emulator and the trivial client implementation correctly implement the VSA abstraction: Let A be the abstract VSA model, and let S be the implementation. Then $traces(S) \subseteq traces(A)$.

D. Self-stabilization

The implementation described here has been extended to be self-stabilizing, guaranteeing that despite possibly arbitrary initial states of real nodes in a VSA's region, the real nodes eventually converge to properly emulate the VSA. To do this, the implementation described above is extended with several trivial local checking and correction actions, as well as a rule that if a broadcast is received at a process p indicating a different head of the $guards$ vector than p has, then p quits emulating the VSA and tries to rejoin the emulation. This rule is an important one, helping us guarantee convergence of emulators to one consistent emulation state rather than competing versions.

Locally checked/corrected variables. In the implementation in Figure 8 we did not describe the local correction actions that clients should perform when elements of their state are obviously corrupted. Rather than write explicit actions for local correction, we describe them briefly here.

There are several local state configurations that indicate to a client $\text{VSAE}_{u,p}$ that its state is one that could not have occurred unless it had been corrupted. These configurations are:

- $status =$ leader and $\langle p, joinreqts\rangle \neq$ head($guards$)
- $status =$ guard and $\langle p, joinreqts\rangle \notin guards$
- $joinreqts > now$
- $round > timeslice + k \cdot t_{slice} + d$
- $timeslice \neq$ nextTS(now) or nextTS(now)$-t_{slice}$
- $\exists m \in (procedq \cup simq) : m.ts > now - d$
- $\exists\langle q, ts\rangle \in joinreqs : ts > now - d$
- $\exists m \in simq : m.ts < now - [(k+1)\cdot t_{slice} + 2d]$

In each of these cases, the client sets its $status$ to `startjoin` to clear its variables and try to re-join.

There are also configurations that indicate that a corruption or failure has occurred, though not necessarily at client $\text{VSAE}_{u,i}$. In these cases we simply update the variables to remove the inconsistencies:

- If $vstate.now < now - e$ then $vstate.now \leftarrow now - e$
- If $\exists m \in holdq : m.ts > now$ or $m.ts < now - d$ then $holdq \leftarrow holdq/\{m\}$
- If $\exists\langle q, ts\rangle \in guards : ts > now - d$ then $guards \leftarrow guards/\{\langle q, ts\rangle\}$

Correctness of self-stabilization. Consider those processes with $reg = u$ and an execution starting from a time t that is ϵ_{sample} time after no additional corruption failures occur. The following then hold:

- Consider a region u and a timeslice ts. If exactly one process broadcasts emulation state for region u in timeslice ts, then by d time into the next timeslice, all processes with $status =$`guard` will have the same values for $vstate$ and $guards$.
- Consider a region u and a timeslice ts. If more than one process broadcasts emulation state for region u in timeslice ts then by d time into the next timeslice, all processes in the region will have $status =$`startjoin` or `trying`.
- Consider a region u and timeslice ts such that all processes with $reg = u$ have $status =$`trying` or `startjoin`. Consider the subset S of these processes that are alive in the region through d time after the start of the $ts + k + 1^{st}$ timeslice. Order the members of S by process id. If at least one of the first k processes in S remains alive in the region through the next k timeslices, then by the end of the $ts + 2k + 1^{st}$ timeslice a message with attached emulation state will be broadcast by exactly one process.
- Eventually, every process with $status =$`guard` is in $guards$.
- By d time after t, any messages added to $simq$ were actually sent and any join requests in $joinreqs$ were actually sent.
- Consider the case where the emulation state is sent by one process at least d time after t in some timeslice ts. By d time into timeslice $ts + 1$, all processes with $status =$`guard` will have the same $simq$ and $joinreqs$.

These together imply that eventually the emulation of the VSA converges so that all guard processes in the emulation share a consistent view of the emulation state.

V. Current and Future Work

Here we describe some current and future work for the VSA layer, including the examination of more realistic system models, consideration of more efficient implementations, and design of applications for the VSA layer.

A. Model extensions and implementation optimizations

The system model assumed here makes optimistic assumptions about clock synchronization and accurate region knowledge that we are addressing. We are also working on several other model extensions and implementation optimizations. There is current work in simulating and implementing this layer in more realistic system models that we hope will help guide improvements and realistic implementations of this layer.

Incorporating collisions. Our implementation should be extended to a more realistic communication model that allows message collisions. In particular, consider the availability of four channels per region in the network, provided either through frequency allocation or additional timeslicing.

The $guards$ vector used to maintain consistency of the emulation of the VSA defines an orderly timeslicing of one communication channel. This channel is dedicated to use by the current leader of the $guards$ vector. Since in normal operation, communication on this channel results in transmissions by at most process per timeslice, any collisions on this channel are treated as errors that result in processes in the region re-joining.

For the other three channels, it is convenient to consider *consensus channels*, a communication channel abstraction in networks with collisions. If a collision occurs, the channel produces one winning message that is succesfully transmitted, representing the result of a successful back-off protocol or completion of an execution of consensus.

We dedicate one consensus channel each for join and restart requests and for client-to-VSA communication. The implementation described here is modified slightly to incorporate extra delays that may result from having to re-submit transmissions. To be certain that schemes for neighboring regions do not result in collisions with each other, we either further timeslice the communication channel or use different sets of frequencies at neighboring regions.

Leader election alternatives. The bulk of the implementation presented in this paper consists of performing a simple leader election. We are separating the leader election portion of the algorithm from the the rest of the implementation in order to more easily take advantage of superior region-based leader election algorithms for mobile networks. These leader election algorithms could be designed to produce stable outputs that take into account factors such as location, speed, power constraints, and reliability of individual nodes in a region.

Implementation optimizations. There are a number of ways in which we can optimize the current VSA implementation for various network scenarios and applications. One simple optimization would be to attach message identifiers, rather than whole messages, to the the emulated state being sent in the algorithm. These identifiers are sufficient to allow guards to determine which saved messages can be thrown out. Also, as implemented now, everybody in a region who is not a guard is trying to become one. One might modify the implementation to be more power consumption friendly by not requiring mobile nodes to always attempt to emulate the VSA.

It is also possible to use state replication approaches that are hybrids of the ones presented here and in [8]. For example, to simplify the discussion we are assuming that the transmitting guard (the current leader) transmits its view concerning the guard vector as well as the latest value of the simulated VSA state. The rest of the guards copy the state and use it as the current most updated version of the data on which any queued actions for the VSA are performed. This simple strategy results in simple self-stabilization and correctness proofs, but implies high communication overhead. However, it allows joining guard nodes to be updated instantly and aids in fast stabilization of the VSA after corruption faults. Optimizations are possible to avoid sending identical shared data if these issues are relatively unimportant; for example we can repeatedly use a random key and hash function that verifies with high probability that the data is identical and transmits the data only when required, or we can allow guards to independently maintain the replicated state in parallel by determinizing the abstract machine being emulated and ensuring all guards receive the same input messages.

B. Applications for the VSA layer

We believe that the VSA layer can be very useful in a number of applications, including some of the more difficult coordination applications for nonhomogenous networks oftentimes desired in true mobile ad hoc deployments. In this section we list several applications that would benefit from the VSA abstraction. We start with basic communication primitives and then go on to describe some more complicated applications.

VSA to VSA communication. One important application would be a means by which remote VSAs can communicate. To implement this service, we would program VSAs to utilize the fixed tiling of the network to forward messages to other VSAs. A message would be forwarded from the source VSA to the destination VSA along a path of neighboring VSAs.

Each VSA chooses a neighboring VSA to forward the message to. The choice of a particular neighbor may be made according to the criteria of shortest path to the destination or greedy DFS as suggested in [10]. Here, however, we would have the advantage of a fixed tiling, rather than the ad-hoc imaginary tiling used in that algorithm. Retransmissions along greedy DFS explored links may be used to cope with repeated crashes and recoveries [11]. The GOAFR algorithm [16], combining greedy routing and face routing, can also be used to give efficient routing in the face of "holes" in the VSA tiling.

Location management. Location management is a complicated task to achieve in Ad-Hoc networks. The VSA abstraction associates (virtual) memory and actions (virtual automata) to fixed geographic regions. We can use one VSA for each client which serves the client as its *home location*. The home location VSA is responsible for maintaining location information for the mobile client. Whenever a client p would like to locate/-communicate with another client q, p uses the result of (a predefined global) hash function on the identifier of q for computing the region identifier of the VSA that serves as the home location for q. In order to ensure a more robust scheme that tolerates deserted/temporarily-crashed virtual automata, the above basic scheme is extended in [11] to use several VSAs as the home locations of a mobile client. In this case the hash function returns a sequence of region identifiers used to update location information and to support queries concerning location information.

Population attribute directories. Location management schemes may be extended to support queries for mobile clients with special characteristics. One example would be to search for a medical doctor in an area during an emergency. The VSA abstraction serves such applications well by recording attributes of clients at VSAs. When a query for a certain client type arrives, the VSA checks its record (and possibly its neighboring VSAs) for clients matching the query and responds.

HikerNet database. VSAs that correspond to geographical locations of interest like a mountain top, campsite in a forest, or riverside picnic area could be used by hikers as a source of on-line information. It is oftentimes infeasible to have a fixed computer station at these regions. However, transient occupancy by hikers on popular trails, on good hiking days, should be enough to maintain VSAs and connectivity. A VSA could maintain a database of summary information about its own local conditions such as temperature, wind speed, and the number of hikers in its area. It could also be extended to maintain a message board of comments such as “the river is impassable” or “a dangerous animal is nearby.” This database can then be queried by hikers curious about conditions in the area.

The database information could be maintained in a history format. At any time, from anywhere in the area of the network of VSAs, someone can query using the VSA-to-VSA communication service to get recent information about a designated location. Regions can become unoccupied, in which case the history disappears and starts over when new people arrive. The history will be complete for as long as a VSA is maintained by continuing occupancy of the forest location.

Some resiliency can be built in by automatically keeping copies of histories backed up at neighboring VSAs. In addition, the collected information could be sent to a central, reliable, known location by a background convergecast algorithm that is executed by the VSA network. This backup concept is useful in general for a number of database applications.

Virtual fence/ Virtual border control. The Ad-Hoc nature of a system is not necessarily due to mobility. Oftentimes new sensors are deployed in an area to restore sensor density after failure of some sensors. The VSA abstraction is useful in handling such changes. A “fence” of VSAs could be useful in this case for applications such as tracking and summarizing events, as well as triggering particular response actions such as “report to command and control” or “light the beacon.”

Hierarchical distributed data structures. In large deployments it can be desirable to establish a multi-layer hierarchy in the network. Hierarchies are used in a variety of algorithms in order to guarantee attractive locality properties. We consider overlaying an tree on the VSA regions. These trees could, for example, be used to allow clients to register attributes with various nodes. Other clients can query the attribute tree to find collections of nodes that have some set of attributes. These queries can be designed to return local answers.

Acknowledgements. We would like to thank Rui Fan for providing helpful comments on drafts of this paper.

REFERENCES

[1] *ACM Transactions on Sensor Networks*.
[2] *Ad Hoc Networks Journal*, Elsevier.
[3] Akylidz, I.F., Su, W., Sankarasubramanian, Y., and Cayirci, E., “Wireless sensor networks: a survey”, *Computer Networks* (Elsevier), 38(4), pp. 393–422, 2002.
[4] Camp, T., Liu, Y., “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.
[5] Demers, A., Gehrke, J., Rajaraman, R., Trigoni, N., and Yao, Y., “Energy-Efficient Data Management for Sensor-Networks: A Work-In-Progress Report”, *2nd IEEE Upstate New York Workshop on Sensor Networks*, `comlab.ecs.syr/edu/workshop`, 2003.
[6] Dijkstra, E.W., “Self stabilizing systems in spite of distributed control”, *Communications of the ACM*, vol. 17, pp. 643–644, 1974.
[7] Dolev, S., *Self-Stabilization*, MIT Press, 2000.
[8] Dolev, S., Gilbert, S., Lynch, N., Schiller, E., Shvartsman, A., and Welch, J., “Virtual Mobile Nodes for Mobile Ad Hoc Networks”, *International Conference on Principles of Distributed Computing (DISC)*, 2004. Also Brief announcement in *Proceedings of the 23th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2004.
[9] Dolev, S., Gilbert, S., Lynch, N., Shvartsman, A., Welch, J., “GeoQuorums: Implementing Atomic Memory in Ad Hoc Networks”, *17th International Conference on Principles of Distributed Computing (DISC)*, Springer-Verlag LNCS:2848, pp. 306-320, 2003.
[10] Dolev, S., Herman, T., and Lahiani, L., “Polygonal Broadcast, Secret Maturity and the Firing Sensors”, *Third International Conference on Fun with Algorithms (FUN)*, pp. 41-52, May 2004. Also Brief announcement in *Proceedings of the 23th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2004.
[11] Dolev, S., Lahiani, L., Lynch, N., Nolte, T., “Self-Stabilizing Mobile-Sensor Home Location Management”, Manuscript, 2004.

[12] Hubaux, J.P., Le Boudec, J.Y., Giordano, S., and Hamdi, M., "The Terminodes Project: Towards Mobile Ad-Hoc WAN", *Proceedings of MOMUC*, 1999.

[13] *IEEE Pervasive Computing: Mobile and Ubiquitous Systems*.

[14] *IEEE Transactions on Mobile Computing*.

[15] Kaynar, D., Lynch, N., Segala, R., and Vaandrager, F., "The Theory of Timed I/O Automata", Technical Report MIT-LCS-TR-917a, MIT Laboratory for Computer Science, Cambridge, MA, 2004.

[16] Kuhn, F., Wattenhofer, R., Zhang, Y., Zollinger, A., "Geometric Ad-Hoc Routing: Of Theory and Practice", *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2003.

[17] Li, J., Jannotti, J., De Couto, D.S.J., Karger, D.R., and Morris, R., "A Scalable Location Service for Geographic Ad Hoc Routing", *Proceedings of Mobicom*, 2000.

[18] Nath, B., Niculescu, D., "Routing on a curve", *ACM SIGCOMM Computer Communication Review*, 33(1), pp. 150 – 160, 2003.

[19] Navas, J.C., Imielinski, T., "Geocast – geographic addressing and routing", *Proceedings of the 3rd MobiCom*, 1997.

[20] Ratnasamy, S., Karp, B., Yin, L., Yu, F., Estrin, D., Govindan, R., and Shenker, S., "GHT: A Geographic Hash Table for Data-Centric Storage", *First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2002.