Timed Virtual Stationary Automata for Mobile Networks

Shlomi Dolev[1], Seth Gilbert[2], Limor Lahiani[1], Nancy Lynch[2], and Tina Nolte[2]

[1] Dep. of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
[2] MIT CSAIL, Cambridge, MA 02139, USA

Abstract. We define a programming abstraction for mobile networks called the *Timed Virtual Stationary Automata* programming layer, consisting of 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 timed VSA using the real mobile nodes that are currently residing in the VSA's region. We also discuss examples of applications whose implementations benefit from the simplicity obtained through use of the VSA abstraction.

Keywords: Ad-hoc networks, mobile computing, location-aware distributed computing, fault tolerance/availability, virtual infrastructure, state replication, virtual machine

1 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. Here we focus on mobile ad-hoc networks, where mobile processors attempt to coordinate despite minimal infrastructure support. This paper develops new techniques to cope with this dynamic, heterogeneous, and chaotic environment.

We mask the unpredictable behavior of mobile networks by defining and emulating a *virtual* infrastructure, consisting of *timing-aware* and *location-aware* machines at fixed locations, that mobile nodes can interact with. The static virtual infrastructure allows application developers to use simpler algorithms — including many previously developed for fixed networks.

There are a number of prior papers that take advantage of geography to facilitate the coordination of mobile nodes. For example, the GeoCast algorithms [1, 19],

[1] Partially supported by IBM faculty award, NSF grant and the Rita Altura chair in computer sciences. Email: {`dolev, lahiani`}`@cs.bgu.ac.il`.

[2] 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`.

GOAFR [13], 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. Other papers [10, 14, 21] 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. We suggest a more systematic approach; many algorithms presented in these papers could be simplified by using a fixed, predictable timing-enabled infrastructure.

In industry there have been a number of attempts to provide specialized applications for ad-hoc networks by organizing some sort of virtual infrastructure over the mobile nodes. PacketHop and Motorola envision mobile devices cooperating to form mesh networks to provide communication in areas with wireless-broadcast devices but little fixed infrastructure [15,26]. These virtual infrastructures could allow on-the-fly network formation that can be used at disaster sites, or areas where fixed infrastructure does not exist or has been damaged. BMW and other car manufacturers are developing systems that allow cars to communicate about local road or car conditions, aiding in accident avoidance [11, 17, 22, 25].

Each of the above examples tackles very specific problems, like routing or distribution of sensor data. A more general-purpose virtual infrastructure, that organizes mobile nodes into general programmable entities, can make a richer set of applications easier to provide. For example, with the advent of autonomous combat drones [24], the complexity of algorithms coordinating the drones can make it difficult to provide assurance to an understandably concerned public that these firepower-equipped autonomous units are coordinating properly. With a formal model of a general and easy-to-understand virtual infrastructure available, it would be easier to both provide and prove correct algorithms for performing sophisticated coordination tasks.

Virtual Stationary Automata programming layer. The programming abstraction we introduce in this paper consists of a static infrastructure of fixed, timed virtual machines with an explicit notion of real-time, called *Virtual Stationary Automata* (VSAs), distributed at known locations over the plane, and emulated by the real mobile nodes in the system. 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. This programming layer provides mobile nodes with a virtual infrastructure with which to coordinate their actions. Many practical algorithms depend significantly on timing, and many mobile nodes have access to reasonably synchronized clocks. In the VSA programming layer, the virtual automata also have access to *virtual* clocks, guaranteed to not drift too far from real-time. These virtual automata can then run programs whose behaviour might be dependent on the continuous evolution of timing variables.

Our virtual infrastructure differs in key ways from others that have previously been proposed for mobile ad-hoc networks. The GeoQuorums algorithm [6, 7] was the first to use virtual nodes; the virtual nodes in that work are atomic objects at fixed geographical locations. More general virtual mobile automata were suggested in [5]; our automata are stationary, and are arranged in a connected pattern that is similar to a traditional wired network. Our automata also have more powerful computational capabilities than those in [5] in that ours include timing capabilities, which are important for many applications. Finally, we use a different implementation stategy for virtual nodes than in [5], incurring less communication cost and enabling us to provide virtual clocks that are never far from real-time.

Emulating the virtual infrastructure. Our clock-enabled VSA layer is emulated by the real mobile nodes in the network. Each mobile node is assumed to have access to a GPS service informing it of the time and region it is currently in. A VSA for a geographic region is then emulated by a subset of the mobile nodes populating its region: the VSA state is maintained in the memory of the real nodes emulating it, and the real nodes perform VSA actions on behalf of the VSA. The emulation is shared by the nodes while one leader node is responsible for performing the outputs of the VSA and keeping the other emulators consistent. If no mobile nodes are in the region, the VSA fails; if mobile nodes later arrive, the VSA restarts.

An important property of our implementation is that it is self-stabilizing. Self-stabilization [3,4] 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 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.

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 motion coordination, tracking, traffic management, and traffic coordination. The key idea in all cases is to locate data and computation at timed VSAs throughout the network, thus relying on the virtual infrastructure to simplify coordination in ad-hoc networks. This infrastructure can be used to implement services such as routing that are oftentimes thought of as the lowest-level services in a network.

2 Datatypes and system model

The system consists of a finite collection of mobile client processes moving in a closed, connected, and bounded region of the 2D plane called R. Region R is partitioned into predetermined connected subregions called *tiles* or *regions*, labeled with unique ids from the set of tile identifiers U. In practice it may be convenient to restrict tiles to be regular polygons such as squares or hexagons. We define a neighbor relation $nbrs$ on ids from U: two tiles u and v are neighbors iff the supremum distance between points in $tile(u)$ and $tile(v)$ is bounded by a constant r_{virt}.

Each mobile node C_p, $p \in P$, the set of mobile node ids, is modeled as a mobile timed I/O automaton whose location in R at any time is referred to as $loc(p)$. Mobile node speed is bounded by a constant v_{max}. We assume each node occasionally receives information about the time and its current region u; a $\mathsf{GPSupdate}(u, now)_p$ happens every ϵ_{sample} time. While GPS is not entirely accurate in reality, as long as an error bound is known, its effects here are small. We assume the node's local clock now progresses at the rate of real-time.

Each client is equipped with a local broadcast communication service called P-bcast, with a minimum broadcast radius of r_{real} and a message delay d. This service allows each client C_p to broadcast a message to all nearby clients through $\mathsf{bcast}(m)_p$ and receive messages broadcast by other clients through $\mathsf{brcv}(m)_p$ actions. We assume that a local broadcast service guarantees two properties: integrity and reliable local delivery. *Integrity* guarantees that every message received was previously broadcast. *Reliable local delivery* (roughly) guarantees that a transmission will be received by nearby nodes: If client C_p broadcasts a message, then every client C_q within r_{real} distance of C_p's transmission location during the transmission interval of length d receives the message before the end of the interval.

Clients are susceptible to stopping and corruption failures. After a stopping failure, a client performs no additional local steps until restarted. If restarted, it starts operating again from an initial state. If a node is corrupted, it suffers from a nondeterministic change to its program state.

Additional arbitrary external interface actions and local state used by algorithms running at the client are allowed. For simplicity local steps are assumed to take no time.

3 Virtual Stationary Automata programming layer

Here we describe the *Virtual Stationary Automata* programming layer. This abstraction includes the real mobile nodes discussed in the last section, the virtual stationary automata (VSAs) that the real nodes emulate, and a local broadcast service, V-bcast, between them (see Figure 1). The layer allows developers to write programs for both mobile clients and stationary tiles of the network as though broadcast-equipped virtual machines exist in those tiles. We begin by describing the properties of VSAs and then describe the V-bcast service.

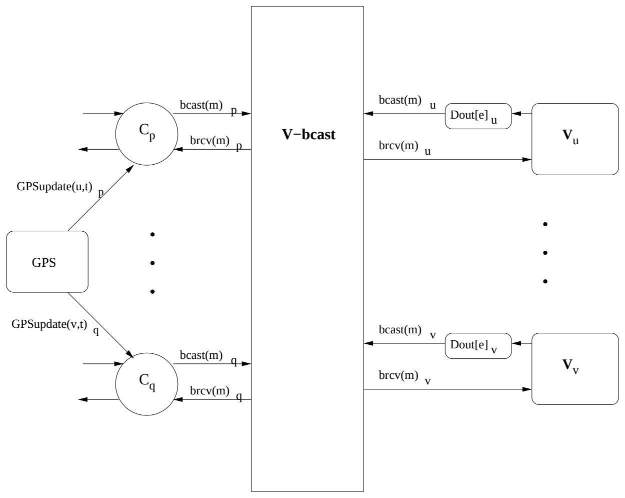

Fig. 1. Virtual Stationary Automata abstraction. VSAs and clients communicate using the V-bcast service. VSA bcasts may be delayed in Dout buffers.

3.1 Virtual Stationary Automata

An abstract VSA is a timing-capable virtual machine. We formally describe such a timed machine for a tile u, V_u, as a TIOA whose program can be referred to as a tuple of its action signature, sig_u, valid states, $states_u$, a start state function, $start_u$, mapping clock values to appropriate start states, a discrete transition function, δ_u, and a set of valid trajectories of the machine, τ_u. Trajectories [12] describe state evolution over intervals of time.

A virtual automaton V_u's external interface is restricted to include only stopping failure, corruption, and restart inputs and the ability to broadcast and receive messages (the restriction guarantees the VSA can be emulated by mobile nodes). Corruptions result in a nondeterministic change to any portion of V_u's state, $vstate$, including the virtual clock $vstate.now$. As with mobile clients, this now value is assumed to progress at the rate of real-time and, outside of failure, equal real-time. Since a VSA is emulated by physical nodes (corresponding to clients) in its region, its failures are defined in terms of client movements and failures in its region: (1) If no clients are in the region, the VSA is crashed, (2) If V_u is failed but a client C_p enters the region and remains for at least $t_{restart}$ time, then in that interval of time V_u restarts, (3) If no client failure (corruption or stopping) occurs in an alive VSA's region over some interval, the VSA does not suffer a failure during that interval, and (4) A VSA may suffer a corruption only if a mobile client in its region suffers a corruption; our self-stabilizing implementation of a VSA guarantees that starting from an arbitrary configuration of the emulation, the emulation's external trace will eventually look like that of the abstract VSA, starting from a corrupted abstract state.

3.2 V-bcast service

The V-bcast service is a "virtual" broadcast communication service with transmission radius r_{virt}. It is similar to that of the real nodes' P-bcast service and implemented using the P-bcast service. It allows broadcast communication between neighboring VSAs, between VSAs and nearby clients, and between clients through bcast and brcv actions, as before. V-bcast guarantees the integrity property described for P-bcast, as well as a similar reliable local delivery property. The *reliable local delivery* property for V-bcast is as follows: If a client or VSA in a region u transmits a message, then every client or VSA in region u or neighboring regions during the entire time interval starting at transmission and ending d later receives the message by the end of the interval. (For this definition, due to GPSupdate lag, a client is still said to be "in" region u even if it has just left region u but has not yet received a GPSupdate with the change.)

Notice that V-bcast's broadcast radius is different from that of P-bcast; since virtual broadcasts are performed using real broadcasts, the virtual transmission radius cannot be larger than the real. Recall r_{virt} is the supremum distance between points in two neighboring tiles. V-bcast then allows a real node p and a VSA for tile u to communicate as long as the node is at most r_{virt} distance from any point in tile u and a VSA to communicate with another VSA as long as they are in neighboring tiles. The implementation of the V-bcast service using the mobile clients' P-bcast service introduces the requirement that $r_{virt} \leq r_{real} - 2\epsilon_{sample} \cdot v_{max}$. The $2\epsilon_{sample} \cdot v_{max}$ adjustment guarantees that two nodes emulating VSAs for tiles they have just left (because they have not yet received GPSupdates that they've change tiles) can still receive messages transmitted to each other. If GPS error is considered, we would compensate by further decreasing r_{virt} by twice the error bound.

3.3 Delay augmentation

The overhead of emulating V_u may introduce additional delays in the broadcasting of messages. 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 are represented with a buffer Dout$[e]_u$, composed with V_u's bcast output. The buffer delays delivery of messages by some nondeterminstic time $[0, e]$. Program actions of V_u must be written taking into account the emulation parameter e, just as it must the message delay factor d. A discussion of the value of e is in Section 4.4.

4 Implementation of the VSA layer

We describe the implementation of a VSA by mobile clients in its tile in the network. At a high level, the individual mobile clients in a tile share emulation of the virtual machine through a deterministic state replication algorithm while also being coordinated by a leader. We begin by describing a totally-ordered broadcast service and leader election service for individual regions, also implemented using the underlying real mobile nodes, that we will use in our replication algorithm. We then focus on describing the core emulation algorithm, briefly sketch correctness, and analyze emulation overhead.

4.1 TOBcast service

In order to keep emulators' state consistent, emulators must process the same sets of messages in the same order. We accomplish this by using the emulators' clocks and P-bcast service to implement a TOBcast service for each region and client. This service allows a client C_p in tile u to broadcast m, TOBcast$(m)_{u,p}$, and to have the message be received, TOBrcv$(m, u)_{v,q}$, by clients in $tile(u)$ and neighboring tiles exactly d time later. To implement this service, when a client wants to TOBcast m from itself or its tile, it tags m with its current tile, time, message sequence number (incremented when the client sends multiple messages at once), and the client id, and broadcasts it using P-bcast. When a client receives such a message from a client in its tile or a neighboring tile it holds the message in a queue until exactly d time has passed since the message's timestamp. Messages that are exactly d old are then TOBrcved in order of sender id and sequence number, ordering the messages. Timestamps are also used to ensure self-stabilization; this is similar to the use of GPS oracles in [9]. To avoid the use of shared variables, we include input and output actions so the TOBcast service can inform the client whether all messages sent up to d time ago have been received. Most complications in the use of these actions come from self-stabilization.

4.2 Leader election service

Here we describe the specifi cation for a leader election service required for our emulator implementation. We divide time up into segments of length t_{slice} called timeslices, that begin on multiples of t_{slice}. Assume $t_{slice} \geq 4d$. When there are no corruption failures, the leader election service for a region u guarantees:
(1) There is at most one leader of a region at a time, and the leader is in the region

(or within $\epsilon_{sample} \cdot v_{max}$) distance,

(2) If a process p becomes leader of region u at some time, then at that time either:
  (a) there was a prior leader of region u during an interval starting at least d after p entered u and ending after some multiple of t_{slice} at least $2d$ later, or
  (b) there is no process in u where a prior leader such as in (a) can be found,

(3) If a process ceases being leader at time t then it will be at least d time before a new leader is chosen,

(4) For any two consecutive timeslices such that at least one process is alive in u for both timeslices and no failures occur in the latter timeslice, there will be a leader in one of the two timeslices from at least $2d$ time before the end of the timeslice to the end of the timeslice.

Property (2) guarantees that either the process that is chosen as a leader has been in the region long enough to have interacted with a prior leader, or there are no processes for which that is true. Property (3) provides a time gap between leaders that will later be useful in guaranteeing that a new leader had heard all prior leader broadcasts before it became a leader.

One example of a self-stabilizing heartbeat implementation of this leader election specification is as follows: if a process is leader, it broadcasts a leaderhb message every t_{slice} amount of time. Once it fails or leaves the tile, the other processes in the region will synchronously timeout the heartbeat and send restart messages, from which the lowest id process that had previously heard a heartbeat from the leader at least $3d$ time after entering the tile is chosen as leader; this ensures that property (2a) holds. If there is no such process, then the lowest id process becomes leader. This simplistic strategy ignores issues of network contention or power management. We briefly discuss alternative leader election strategies in Section 6.

4.3 Emulator implementation

Here we describe a fault-tolerant implementation of a VSA emulator. We first describe how our leader-based emulation generally works and then address details in the emulation. The signature, state, and trajectories for the algorithm are in Figure 2 and the actions are in Figure 3. Line numbers refer to lines in Figure 3.

Leader-based virtual machine emulation. In our virtual machine emulation, at most one of the mobile nodes in a VSA's tile is a leader (chosen by the leader election service), with primary responsibility for emulating the VSA and performing VSA outputs. A leader stores and updates the state of the VSA (including the VSA's clock value) locally, simulating all actions of the VSA based on it. When the leader receives a TOBcast message, it places the message in a local saved message queue (lines 33-37) from which it simulates the VSA brcving (processing) the message (lines 39-45). If the VSA is to perform a local action, the leader simulates its effect on the VSA state (lines 47-54). If the VSA action is to bcast a message, the leader places the message in an outgoing VSA queue (lines 53-54), to be removed and TOBcasted with the tile as the source by the leader, in the VSA's stead (lines 56-61).

For fault-tolerance and load balancing reasons, it is necessary to have more than just the leader maintaining a VSA. In our multiple emulator approach, a VSA is maintained by several emulators, including at most one leader, each maintaining and updating its

```
   Signature:
2    Input GPSupdate$(v, t)_p$, $v \in U$, $t \in \mathbb{R}$
     Input leader$(val)_{u,p}$, $val \in Bool$
4    Input TOBnext$(t)_{u,p}$, $t \in \mathbb{R}$
     Input TOBrcv$(m, v)_{u,p}$, $v \in \{u\} \cup nbrs(u)$
6    Output TOBprobe$_{u,p}$
     Output TOBcast$(m)_{u,p}$, $m \in (Msg \times \mathbb{R}) \cup \{$join$\} \cup$
8    $(\{$update$\} \times states_u) \cup (\{$check$\} \times (hash \times N) \times Bool)$
     Internal VSArcv$(m)_{u,p}$
10   Internal VSAlocal$(act)_{u,p}$, $act \in$ internal, output $sig_u$
     Internal correctqueues$_{u,p}$
12   Internal checksum$_{u,p}$

14 State:
     analog $now \in \mathbb{R}$, current real time
16   $reg \in U$, current reg, initially $\perp$
     $nextrcv, joinTS, leadTS, joinreq \in \mathbb{R}$
18   $vstate \in states_u$
     oldsavedq, savedq, outq, queues of msg, timestamp pairs
20   checksum, triple of hashed $V_u$ state, a natural, and a bool

   Trajectories:                                                    22
     satisfies
       $d(now) = 1$                                                 24
       constant reg, joinTS, joinreq, oldsavedq, savedq,
                outq, nextrcv, leadTS, checksum                     26
       $\tau(now).vstate = \tau_u(\tau(now).vstate.now)$
       if $(vstate \neq \perp \wedge vstate.now \geq now - d)$ then  28
         if $vstate.now < now$ then
           $d(vstate.now) = x$, $x > 2$                             30
         else $vstate.now = now$
       else constant vstate                                         32
     stops when
       Any precondition is satisfied.                               34
```

Fig. 2. VSAE$_{u,p}$, emulator at p of V$_u = \langle sig_u, states_u, start_u, \delta_u, \tau_u \rangle$ - signature, state, trajectories.

local copy of the VSA state and saved message queue as above. However, non-leader emulators, unlike leaders, do not transmit messages for the VSA from their outgoing VSA queues, preventing multiple transmission of messages from the VSA. To keep emulators consistent, the emulation trajectories are based on a determinized version of the VSA trajectories.

Emulation details. There are several complications in VSA emulation that arise due to both message delays and process failure:

Joining: When a node discovers it is in a new region, it TOBcasts a join message (lines 23-31). Any process that receives this message stores the timestamp of the message as the latest join request (lines 63-65). If a leader has processed all messages in its saved message queue and TOBcasted all messages in its outgoing VSA queue, it answers outstanding join requests by TOBcasting an update message, containing a copy of the leader's current emulated VSA state (lines 67-75). The leader holds off on performing any additional VSA-related transmissions until it receives this message (line 75). When any process that has been in the region at least $2d$ time receives the update, it adopts the attached VSA state as its own local VSA state and erases its outgoing VSA queue (lines 77-91). (If it has not been in the region $2d$ time, its saved message queue may not have all messages that were too recent to be reflected in the update.)

Catching up to real time: After receipt of an update message, the VSA's clock (and state) can be d behind real time. Intuitively, the VSA emulation is "set back" whenever an update message is received. To guarantee the VSA emulation satisfies the specifications from Section 3 (bounding the time the output trace of the emulation may be behind that of the VSA being emulated), the virtual clock must catch up to real time. This is done by having the virtual clock advance more than twice as fast as real time until both are equal, after which they increase at the same rate. This is formally described in Figure 2, lines 28-32. To guarantee that the virtual clock can catch up before d time,

```
Output TOBprobe$_{u,p}$
Precondition:
    $nextrcv \leq now - d$

Input TOBnext$(t)_{u,p}$
Effect:
    $nextrcv \leftarrow t$

Input GPSupdate$(v, t)_p$
Effect:
    $now \leftarrow t$
    if $reg \neq v$ then
        $reg \leftarrow v$
        $joinTS \leftarrow \infty$

Input leader$(val)_{u,p}$
Effect:
    if $(!\, val \vee joinTS > now - d)$ then
        $leadTS \leftarrow \infty$
    else if $leadTS > now + d$ then
        $leadTS \leftarrow now$

Output TOBcast(join)$_{u,p}$
Precondition:
    $reg = u \wedge joinTS > now$
Effect:
    $joinTS \leftarrow now$
    $nextrcv \leftarrow now - d$
    $leadTS, joinreq \leftarrow \infty$
    $savedq, oldsavedq, outq \leftarrow \emptyset$
    $vstate, checksum \leftarrow \bot$

Input TOBrcv$(m,s)_{u,p}$, $m$.first $\notin$ {check,update,join}
Effect:
    $savedq \leftarrow$ append$(savedq, \langle m.$first$, now - d\rangle)$
    if $(s = u \wedge \exists\, x,y$:[$outq =$ append(append$(x, m), y)$ ]$)$
        then $outq \leftarrow y$

Internal VSArcv$(m)_{u,p}$
Precondition:
    $vstate \neq \bot \wedge \langle m, t\rangle =$ head$(savedq)$
Effect:
    $vstate \leftarrow \delta_u(vstate,$ brcv$(m))$
    $oldsavedq \leftarrow$ append$(oldsavedq,$ head$(savedq))$
    $savedq \leftarrow$ tail$(savedq)$

Internal VSAlocal$(act)_{u,p}$
Precondition:
    $vstate \neq \bot \neq \delta_u(vstate, act) \wedge savedq = \emptyset$
    $nextrcv > now - d \wedge act =$ next$(vstate, \delta_u)$
Effect:
    $vstate \leftarrow \delta_u(vstate, act)$
    if $act =$ bcast$(m)$ then
        $outq \leftarrow$ append$(outq, \langle m, vstate.now\rangle)$

Output TOBcast$(m)_{u,p}$
Precondition:
    $reg = u \wedge leadTS \leq now < nextrcv + d \wedge vstate \neq \bot$
    $vstate.now \geq now - d \wedge \forall \langle m, t\rangle \in outq$: $t \geq now - e$
    $m =$ head$(outq)$
Effect: $outq \leftarrow$ tail$(outq)$

Input TOBrcv(join, $u)_{u,p}$
Effect:
    $joinreq \leftarrow now - d$

Output TOBcast$(\langle$update, $vstate'\rangle)_{u,p}$
Precondition:
    $reg = u \wedge leadTS \leq now < nextrcv + d$
    $(vstate' = vstate \wedge [vstate = \bot \vee (vstate.now = now$
    $\wedge\, outq = \emptyset = savedq \wedge joinreq \neq \infty)\,]) \vee (vstate' = \bot$
    $\wedge\, [vstate.now < now - d \vee \exists\, \langle m, t\rangle \in outq$: $t < now - e\,])$
Effect:
    $joinreq \leftarrow \infty$
    $leadTS \leftarrow now + d$

Input TOBrcv$(\langle$update, $vstate'\rangle, u)_{u,p}$
Effect:
    if $joinreq \leq now - 2d$ then
        $joinreq \leftarrow \infty$
    if $(joinTS \leq now - 2d \wedge vstate' = \bot)$ then
        $vstate \leftarrow start_u(now)$
        $savedq \leftarrow \emptyset$
    else if $joinTS \leq now - 2d$ then
        if $vstate = \bot$ then
            $oldsavedq \leftarrow \emptyset$
        $vstate \leftarrow vstate'$
        $savedq \leftarrow$ append$(oldsavedq, savedq)$
                                    $- \{\langle m, t\rangle : t \leq now - 2d\}$
    $oldsavedq, outq \leftarrow \emptyset$
    $checksum \leftarrow \bot$

Internal correctqueues$_{u,p}$
Precondition:
    $\exists\, \langle m,t\rangle \in oldsavedq \cup savedq$: $t > now - d$
    $\vee\, \exists\, \langle m,t\rangle \in outq$: $t > now$
Effect:
    $savedq, oldsavedq \mathrel{-}= \{\langle m, t\rangle : t > now - d\}$
    $outq \mathrel{-}= \{\langle m, t\rangle : t > now\}$

Internal checksum$_{u,p}$
Precondition:
    $vstate.now$ mod $ttl_{update} = 0 \wedge nextrcv > now - d$
    $savedq = \emptyset \wedge \forall act \in sig_u$-{brcv$(m)$}:$\delta_u(vstate, act) = \bot$
    $checksum \neq \langle$checksum$(vstate), vstate.now/ttl_{update}, *\rangle$
Effect:
    $checksum \leftarrow$
        $\langle$checksum$(vstate), vstate.now / ttl_{update},$ false$\rangle$
    if $(joinreq \neq \infty \wedge joinreq > now - d)$ then
        $joinreq \leftarrow now - d$

Output TOBcast$(\langle$check, $\langle csum, t\rangle, jr\rangle)_{u,p}$
Precondition:
    $reg = u \wedge leadTS \leq now < nextrcv + d \wedge outq = \emptyset$
    $now + d \leq (t + 1) \cdot ttl_{update}$
    $checksum = \langle csum, t,$ false$\rangle \wedge jr == (joinreq \neq \infty)$
Effect:
    $checksum \leftarrow \langle csum, t,$ true$\rangle$

Input TOBrcv$(\langle$check, $\langle csum', t'\rangle, jr\rangle, u)_{u,p}$
Effect:
    $outq \mathrel{-}= \{\langle m, t\rangle : t \leq t' \cdot t_{slice}\}$
    if $(jr \wedge joinreq = \infty)$ then
        $joinreq \leftarrow now - 2d$
    if $([vstate = \bot \wedge joinTS \leq now - 2d \wedge !\, jr\,]$
        $\vee\, [vstate \neq \bot \wedge checksum \neq \langle csum', t', *\rangle\,])$ then
            $joinTS \leftarrow \infty$
    else $checksum \leftarrow \langle csum', t',$ true$\rangle$
```

Fig. 3. VSAE$_{u,p}$, emulator at p of V$_u$ = $\langle sig_u, states_u, start_u, \delta_u, \tau_u\rangle$ - actions.

we require a leader to only transmit an update message once its virtual clock is caught up to real time (line 70).

Message processing: Messages to be received by the VSA are placed in a saved message queue from which emulators simulate receiving the messages. If an update message is received, setting back the state of the VSA, emulators must be able to resimulate receiving messages that were sent up to d time before the update was sent. In order to guarantee this, whenever an emulator processes a message from the saved message queue for the VSA, it moves the message into an old saved message queue (line 44); if a process receives an update message, it moves all messages in that queue that were received after the update was sent back into its saved message queue to be reprocessed (line 88-89).

Making up leader broadcasts: If a leader is supposed to perform broadcasts on the VSA's behalf, but fails or leaves before sending them, the next leader needs to transmit the messages. Since emulators store outgoing VSA messages in a local outgoing queue, the new leader just transmits any messages stored in its outgoing queue (lines 56-61) and removes them. To prevent messages from being rebroadcast by future leaders, emulators that receive a VSA message broadcast by the leader remove it from their own outgoing queues (lines 36-37).

Restarting a VSA: If a process is leader and has no value for the VSA state or has messages in its outgoing queue with timestamps older than the delay augmentation parameter e, it restarts the emulation. It does this by sending an update message with attached state of $\perp$ and then waiting to receive the message (lines 67-75). When processes that have been in the region $2d$ time receive the message d later, they initialize the VSA state and messaging queues and begin emulating a restarted VSA (lines 77-91).

Self-stabilization. Our implementation is self-stabilizing through the use of local correction and update and checksum messages. The update messages sent by a leader contain state information which overwrites any VSA state information at other emulators, bringing emulators into agreement about VSA state. In the event that join requests do not occur very often, if the virtual clock is divisible by ttl_{update}, the emulators calculate and store a checksum of the VSA state. The leader is then responsible for sending out checksum messages with the attached checksum. Emulators, when they receive this message, compare the attached checksum to the version that they have stored. If the versions differ, they re-join. This ensures that emulators will have state consistent with the leader's.

4.4 Correctness and performance evaluation

Correctness roughly consists of guaranteeing liveness of the emulation under certain circumstances and guaranteeing that emulations of an abstract VSA implement the VSA.

We say a VSA emulation is *failed* if no process in the region has VSA state $vstate \neq \perp$ such that $vstate.now \geq now - d$ and its outgoing queue has no messages with timestamps more than e before real-time.

Assume that as a parameter of the system, there is some positive integer k such that if a process is alive in a region from the beginning of any timeslice t through the end of timeslice $t + k$, then there is at least one timeslice in $t + 1 \ldots t + k$ where no failures or leaves of processes occur in the region. We can then show the following:

Lemma 1. *For any non-failed VSA emulation, VSA outputs are not delayed by more than* $e = (k+1) \cdot t_{slice} - d$ *time, and as long as from the beginning of any timeslice there is at least one alive process in the VSA's region with* $vstate \neq \perp$, $vstate.now \geq now - d$*, and an outgoing queue without messages that are older than* e *that remains alive in the region through the following* k *timeslices, the VSA emulation does not fail or restart.*

Lemma 2. *If a VSA is failed in some timeslice but there is an alive process in the VSA's region from the beginning of the timeslice through the following* k *timeslices, then the VSA will be restarted within* e *time.*

Theorem 1. *The VSA emulator and client implementation (*S*) correctly implement the VSA abstraction (*A*):* $\textit{timed-traces}(S) \subseteq \textit{timed-traces}(A)$.

Proof sketch: We introduce an intermediate layer, and describe a (simple) simulation relation [12] between this layer and the abstract layer. We then describe a simulation relation from our implementation to the intermediate layer. Together, this shows the implementation implements the abstract layer.

The intermediate layer is similar to the abstract layer, except that VSAs may have clocks that are behind real-time and have incoming delay buffers that hold each message bound for the VSA until the VSA's clock passes the message's timestamp. This layer captures the idea that VSA state in the emulation can be behind what the corresponding abstract VSA state would be. A simulation relation is then defined to show that this intermediate layer implements the abstract layer, by relating the state of a VSA, its incoming message buffer, and outgoing message buffer in the intermediate layer to what will be the state of that VSA and its delayed outgoing message buffer in the abstract layer, once its virtual clock equals the current real-time.

We then describe a forward simulation relation between the implementation and the intermediate VSA abstraction for non-failed VSA emulations. There are 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:

(1) For any process where $vstate \neq \perp$, the value of $vstate$ is equivalent to $V_u.vstate$ unless there is an update message in transit, in which case $V_u.vstate$ is equal to the attached state in the update message.

(2) If m is a message either in transit to p or in p's saved message queue, then m is in virtual transmission to u. If there is an update message in transit and m is in p's old saved message queue and if m was sent less than d before the update, then it is in virtual transmission to u.

(3) If m is a message in transit to p and was sent by V_u, then the message is in virtual transmission to p.

(4) If m is a message in the outgoing queue and not currently in transit, and no update message is in transit then m is in Dout$[e]$. □

Message complexity. There are two parts to the message overhead introduced by this algorithm. The first is that of the overhead in normal operation introduced over that of the virtual machine if it was real. This is just one checksum-sized message every ttl_{update} time (used for self-stabilization). The second is that of the overhead from dealing with processes joining the emulation. In this case, when a successful join occurs it results in a broadcast of the VSA state and saved message queue, which could contain as many messages as could be received in d time. If M' is the number of messages that can be received in d time, then the bit overhead of a join is $O(|vstate| + |msg| \cdot M')$.

5 Applications for the VSA layer

We believe the VSA layer will be helpful for many applications, including some of the more difficult coordination problems for nonhomogenous networks oftentimes desired in true mobile ad-hoc deployments. It allows application developers to re-use many algorithms originally designed for the fixed network or base station setting, and to design different services for different regions. Here we list several applications whose implementations would benefit from use of the VSA abstraction.

Geo-routing. One important application is to allow arbitrary regions to communicate. This can be easily implemented by VSAs that utilize the fixed tiling of the network to forward messages [9]. Each VSA chooses a neighboring VSA to forward a message to according to criteria of shortest path to destination or greedy DFS as suggested in [8]. The VSA layer offers a fixed tiled infrastructure to depend on, rather than the ad-hoc imaginary tiling used in that algorithm. Retransmissions along greedy DFS explored links can be used to cope with repeated crashes and recoveries [9]. The GOAFR algorithm [13], combining greedy routing and face routing, can be used to give efficient routing in the face of "holes" in the VSA tiling.

Location management and end-to-end routing. Location management is a difficult task in ad-hoc networks, as many algorithms assume fixed infrastructure and raise difficult-to-analyze concerns about data consistency. However, *home location* algorithms are easily implemented using the VSA layer [9]. Each client's id can be hashed to a set of VSAs (home locations) that would store the client's location. The client would occasionally inform its local VSA of its presence. That local VSA would then inform the client's home locations, using a Geo-routing service, of the region. Anyone searching for the client would have their local VSA query the client's home location VSAs, again using the a Geo-routing service, for the client's location.

The home location service can then be used to provide tracking services or end-to-end communication between individual clients [9]. A message is sent to a client by looking up its location using the home location service and then using Geo-routing to send the message to VSAs close to the returned location. Those VSAs that receive the message broadcast it to local clients for delivery by the intended recipient.

Distributed coordination. VSAs corresponding to geographic regions can be a source of on-line information and coordination, directing mobile clients to help them complete distributed systemwide missions. The virtual infrastructure can make it easier to handle coordination of many clients when tasks are complex. Also, many coordination problems can tolerate a VSA in an empty region failing since such regions have no clients to coordinate. The use of a virtual infrastructure to enable mobile clients to coordinate and equally space themselves along a target curve was recently demonstrated in [16]. The paper provides a simple framework for coordinating client nodes through interaction with virtual nodes. It also demonstrates a simplistic "emulator-aware" approach to maintenance of virtual automata; a VSA makes decisions about target destinations for participating clients based partly on information about local population density in an attempt to keep the VSA alive. The approach could be extended to take into account more client or network factors and even to provide active recruitment, where virtual automata can request emulator aid from distant virtual automata regions.

An example of a timed coordination application is that of a *virtual traffic light*. A VSA for a region corresponding to the intersection of roads in a remote area can provide a virtual traffi c light that keeps the light green in each direction for a specifi c amount of time, providing a substitute for the fi xed infrastructure lacking in the region. The VSA would be emulated by computers on vehicles approaching the intersection. Multiple traffi c VSAs can also coordinate to facilitate optimal movement of mobile clients.

Another coordination application is the Virtual Air-Traffi c Controller [20]. The VSA controller uses detailed knowledge of time in order to plan where and when airborne planes should fly. The burden of regulating lateral separation of aircraft could be allocated in a distributed fashion by VSAs, where VSAs assign local planes different time separations and altitudes based on aircraft type and heading. By devolving some decision-making to aircraft, we can both alleviate ground-based bottlenecks and allow for more local control of flight plans, resulting in optimized routes and better fuel economy [23]. Airspace VSAs are easy to envision, given positioning, long-range communications, and computing resources increasingly available on commercial aircraft.

Data collection and dissemination. A VSA could maintain a summary database of information about its local conditions and those of other regions. Clients could then query their local VSA to get recent information about a location. The history is complete as long as the VSA's tile remains occupied. Resiliency can be built in by using techniques already designed for static but failure-prone networks, such as automatically backing up data at neighboring VSAs or sending data to a central, reliable location by a background convergecast algorithm executed by the VSA network.

Hierarchical distributed data structures. Here, tile size is constrained by the broadcast range of the underlying nodes. An hierarchical emulation of the model, where multiple nodes coordinate to emulate larger tiles, can provide a more general infrastructure. The VSA layer can be a basic building block to implement hierarchies in a network that could, for example, be used to allow clients to register and query attributes.

6 Current and future work

The system model assumed so far abstracts away details of the underlying physical layer in order to clearly describe algorithmic issues. Here we discuss some implementation issues and extensions. We also hope that current work simulating this layer and implementing it will guide improvements in our layer implementation.

Non-synchronized clocks. The VSA layer model and implementation could be extended to allow for a known bound on mobile node clock drift. This results in the addition of incoming message delay buffers for VSAs in the abstract model, in addition to the outgoing ones already present.

Emulation strategies to accommodate message collisions. Our work is being extended to a communication model allowing message collisions [2]. One approach is to relax the physical and VSA layer broadcast models to allow message loss in the presence of contention, but guarantee the VSA emulation is reliable by taking advantage of the fact that leader election effectively defines an orderly timeslicing of a communication channel for at least one process. Consider two channels per tile in the network, provided either through frequency allocation or additional timeslicing. Assuming a leader election service for this setting, whichever process is leader can have one channel to itself, allowing it to perform VSA related broadcasts without interference from other processes. The other channel could be used by nodes trying to communicate with the VSA; message loss on this channel would be possible since there could be contention. The leader can then become the arbiter of which messages are actually received by the VSA, by rebroadcasting received messages; other emulators adopt these as the incoming messages for the VSA. Alternatively, a more state transmission heavy approach could be adopted, where non-leader emulators are passive, and the leader periodically broadcasts up-to-date state to them.

Leader election algorithms. Our emulation algorithm utilizes a basic leader election service with a simple interface. Alternative leader election strategies can be considered. For example, a round-robin strategy can help relieve network congestion. Such a strategy could periodically select a new leader from a k-bounded vector of mobile nodes in a region called $guards$. This is done by defining globally known *timeslices* of length t_{slice} and rotating the $guards$ vector each timeslice, defining revolving responsibility for leadership. Whichever process's id and join timestamp pair is currently at the head of the rotating vector is the leader. Processes trying to join the $guards$ vector are appended to it if there is room while leaders that fail to transmit during their timeslice are subsequently dropped from the vector.

A promising area for further research is into region-based leader election algorithms for mobile networks that are designed to produce stable outputs that take into account factors such as location, speed, power constraints, and reliability of individual nodes. Improved leader election guarantees can lead to improved emulation guarantees.

In addition, a leader election service could be extended to inform client nodes if they should participate in emulation at all. Some clients could be told they are not needed for emulation for some period, allowing them to conserve power.

Extensions to non-homogenous networks. In many cases, there are portions of a deployment area that have fixed infrastructure or sensing capabilities and portions that do not. While the model we introduced here does not take into account the fact that some deployments may have some access to fixed infrastructure, the model in this paper should easily be extended to accommodate these mixed deployments.

References

1. Camp, T. and 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.
2. Chockler, G., Demirbas, M., Gilbert, S., Newport, C., and Nolte, T., "Consensus and Collision Detectors in Wireless Ad Hoc Networks", *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2005.

3. Dijkstra, E.W., "Self stabilizing systems in spite of distributed control", *Communications of the ACM*, 1974.
4. Dolev, S., *Self-Stabilization*, MIT Press, 2000.
5. 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.
6. Dolev, S., Gilbert, S., Lynch, N., Shvartsman, A., and Welch, J., "GeoQuorums: Implementing Atomic Memory in Ad Hoc Networks", *17th International Conference on Principles of Distributed Computing (DISC)*, Springer-Verlag LNCS:2848, 2003.
7. Dolev, S., Gilbert, S., Lynch, N., Shvartsman, A., and Welch, J., "GeoQuorums: Implementing Atomic Memory in Ad Hoc Networks", Technical Report MIT-LCS-TR-900, MIT Laboratory for Computer Science, Cambridge, MA, 02139, 2003.
8. 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 to appear in *Ad Hoc Networks Journal*, Elseiver.
9. Dolev, S., Lahiani, L., Lynch, N., and Nolte, T., "Self-Stabilizing Mobile Node Location Management and Message Routing", Symposium on Self Stabilizing Systems (SSS), 2005.
10. Hubaux, J.P., Le Boudec, J.Y., Giordano, S., and Hamdi, M., "The Terminodes Project: Towards Mobile Ad-Hoc WAN", *Proceedings of MOMUC*, 1999.
11. Kan, M., Pande, R., Vinograd, P., and Garcia-Molina, H., "Event Dissemination in High-Mobility Ad-hoc Networks", Technical Report, 2005.
12. Kaynar, D., Lynch, N., Segala, R., and Vaandrager, F., "The Theory of Timed I/O Automata", Technical Report MIT-LCS-TR-917a, MIT LCS, Cambridge, MA, 2004.
13. Kuhn, F., Wattenhofer, R., Zhang, Y., and Zollinger, A., "Geometric Ad-Hoc Routing: Of Theory and Practice", *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2003.
14. 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.
15. Lok, C., "Instant Networks: Just Add Software", *Technology Review*, June, 2005.
16. Lynch, N., Mitra, S., and Nolte, T., "Motion coordination using virtual nodes", To appear: IEEE Conference on Decision and Control, 2005.
17. Morris, R., Jannotti, J., Kaashoek, F., Li, J., and Decouto, D., "CarNet: A Scalable Ad Hoc Wireless Network System", 9th ACM SIGOPS European Workshop, Kolding, Denmark, September 2000.
18. Nath, B. and Niculescu, D., "Routing on a curve", *ACM SIGCOMM Computer Communication Review*, 2003.
19. Navas, J.C. and Imielinski, T., "Geocast- geographic addressing and routing", *Proceedings of the 3rd MobiCom*, 1997.
20. Neogi, N., "Designing Trustworthy Networked Systems: A Case Study of the National Airspace System", International System Safety Conference, Ottawa, Canada, 2003.
21. 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.
22. Sun, Q., and Garcia-Molina, H., "Using Ad-hoc Inter-vehicle Networks for Regional Alerts", Technical Report, 2004.
23. Talbot, D., "Airborne Networks", *Technology Review*, May, 2005.
24. Talbot, D., "The Ascent of the Robotic Attack Jet", *Technology Review*, March, 2005.
25. Vasek, T., "World Changing Ideas: Germany", *Technology Review*, April, 2005.
26. Woolley, S., "Backwater Broadband", *Forbes*, 2005.