Virtual Mobile Nodes
for Mobile *Ad Hoc* Networks
(Extended Abstract)

Shlomi Dolev[1], Seth Gilbert[2], Nancy A. Lynch[2],
Elad Schiller[1], Alex A. Shvartsman[3,2], and Jennifer L. Welch[4]

[1] Dept. of Computer Science, Ben-Gurion U., {dolev,schiller}@cs.bgu.ac.il
[2] MIT CSAIL, {sethg,lynch}@theory.lcs.mit.edu
[3] Dept. of Computer Science & Eng., U. of Connecticut,alex@theory.lcs.mit.edu
[4] Dept. of Computer Science, Texas A&M University, welch@cs.tamu.edu

Abstract. One of the most significant challenges introduced by mobile networks is coping with the *unpredictable* motion and the *unreliable* behavior of mobile nodes. In this paper, we define the *Virtual Mobile Node Abstraction*, which consists of robust virtual nodes that are both predictable and reliable. We present the *Mobile Point Emulator*, a new algorithm that implements the Virtual Mobile Node Abstraction. This algorithm replicates each virtual node at a constantly changing set of real nodes, modifying the set of replicas as the real nodes move in and out of the path of the virtual node. We show that the Mobile Point Emulator correctly implements a virtual mobile node, and that it is robust as long as the virtual node travels through well-populated areas of the network. The Virtual Mobile Node Abstraction significantly simplifies the design of efficient algorithms for highly dynamic mobile *ad hoc* networks.

1   Introduction

Devising algorithms for mobile networks is hard. In this paper we present the Virtual Mobile Node Abstraction, which can be used to make this process easier.

The key challenge in mobile networks is coping with the completely *unpredictable motion* of the nodes. This complication is unavoidable: the defining feature of a mobile network is that the nodes do, in fact, move. The other main difficulty is the *unpredictable availability* of nodes that continually join and leave the system: nodes may fail and recover, or be turned on and off by the user, or may sometimes choose to sleep and save power.

If mobile nodes were reliable and their motion were predictable, the task of designing algorithms for mobile networks would be significantly simplified.

Moreover, if mobile nodes moved in a programmable way, algorithms could take advantage of the motion, performing even more efficiently than in static networks. This idea is illustrated by Hatzis et al. in [10], which defines the notion of a *compulsory* protocol, one that requires a subset of the mobile nodes to move in a pre-specified manner. They present an efficient compulsory protocol for leader election. The routing protocols of Chatzigiannakis et al. [4] and Li et al. [15] provide further evidence that compulsory protocols are simple and efficient.

Alas, users of mobile devices are not amenable to following instructions as to where their devices may travel. It may be difficult to ensure that mobile nodes move as desired, especially for highly dynamic systems where nodes may fail or be diverted from the prescribed path. Thus our objectives are (*a*) to retain the effectiveness of the compulsory protocols, and (*b*) to achieve this without imposing requirements on the motion of the nodes.

Our Contributions

In this paper we introduce the Virtual Mobile Node (VMN) Abstraction, and show how it can be used to design distributed algorithms for mobile *ad hoc* networks. We develop an algorithm, the *Mobile Point Emulator*, that implements the VMN abstraction, and show that it is correct and efficient.

Virtual Mobile Nodes. We propose executing algorithms on both *virtual mobile nodes* (VMNs), abstract nodes that move in a predetermined, predictable manner, and clients (i.e., real mobile nodes), which move in an unpredictable manner. The motion of a VMN is determined in advance, and is known to the programs executing on the mobile nodes. For example, a VMN may traverse the plane in a regular pattern, or it may perform a pseudorandom walk.

The motion of the virtual nodes may be completely uncorrelated with the motion of the real nodes: even if all the real nodes are moving in one direction, the virtual nodes may travel in the opposite direction. Consider, for example, an application to monitor traffic on a highway: even though all the cars are driving in one direction, a VMN could move in the opposite direction, notifying oncoming cars of the traffic ahead.

A virtual node is prone to "crash-reboot" failures. As long as the virtual node travels through dense areas of the network, the virtual node does not fail. However, if the VMN moves to an empty spot — where there are no mobile nodes to act as replicas — a failure may occur. The VMN can recover to its initial state when it reenters a dense area.

The virtual nodes and the clients communicate using only a local communication service; no long-distance communication is required.

Implementing Virtual Mobile Nodes. We present the *Mobile Point Emulator*, a new algorithm that implements robust VMNs. The main idea of the algorithm is to allow real nodes traveling near the location of a VMN to assist in emulating the VMN. In order to achieve robustness, the algorithm replicates the state of a virtual node at a number of real mobile nodes. As the execution proceeds, the algorithm continually modifies the set of replicas so that they always remain

near the virtual node. We use a replicated state machine approach, augmented to support joins, leaves, and recovery, to maintain the consistency of the replicas.

Other Related Work

While the idea of executing algorithms on virtual mobile nodes was inspired by the development of compulsory protocols [10, 4, 15], many of the techniques used in the Mobile Point Emulator were developed as part of the GeoQuorums algorithm [6, 7], which defines a Focal Point Abstraction in which geographic regions of the network—*focal points*—simulate atomic objects. The Virtual Mobile Node Abstraction differs from the Focal Point Abstraction in four main ways. First, in the earlier work, the focal points are *static*: they are limited to fixed, predetermined locations. In this paper, we implement virtual mobile nodes that *move*, traveling on an arbitrary, predetermined path. Second, the Focal Point Abstraction includes only atomic objects, such as read/write registers. In this paper, the virtual mobile nodes can be arbitrary automata. Third, the focal points cannot recover, should they fail, whereas the VMN Abstraction supports recovery. Fourth, the Focal Point Abstraction uses a GeoCast service, a relatively expensive non-local service, to communicate with clients. In the VMN Abstraction, virtual nodes and clients communicate using only local communication.

This paper also generalizes the *PersistentNode* abstraction by Beal [1, 2]. A PersistentNode is a virtual entity that travels around a static (rather than mobile) sensor network. It can carry with it some state, but implements neither atomic objects (as in GeoQuorums), nor arbitrary automata (as in this paper).

The work of Nath and Niculescu [18] also takes advantage of precalculated paths to forward messages in dense networks. Messages are routed along trajectories, where nodes on the path forward the messages. Similarly, prior GeoCast work (for example, [19, 3]) attempts to route data geographically. In many ways, these strategies are *ad hoc* attempts to emulate some kind of traveling node. We provide a more general framework to take advantage of predictably dense areas of the network to perform arbitrary computation. A significant focus of these prior papers is *determining* good trajectories, a problem that we do not address.

Document Structure

We first describe the underlying system model in Section 2, and present the VMN Abstraction in Section 3. We present the Mobile Point Emulator and the implementation of the VMN Abstraction in Section 4, and sketch a proof of correctness in Section 5. In Section 6, we briefly discuss several simple algorithms that could execute on virtual nodes. Finally, in Section 7, we discuss open problems and future work. For more details, see [5].

2   Basic System Model

The underlying system model consists of real mobile nodes moving in a bounded region of a two-dimensional plane. Each mobile node is assigned a unique identifier from a finite set, I. The real mobile nodes may join and leave the system,

and may fail at any time. (We treat leaves as failures.) The real mobile nodes can move on any continuous path, with speed bounded by a constant, v_{max}.

The *Geosensor* is a component of the environment that maintains the current location of each mobile node. It also maintains the current real time. A mobile node receives geo-update(t, ℓ) updates from the Geosensor, notifying it of the current time and its current location. Throughout this paper, we assume exact knowledge of the time and location; in fact, all the algorithms presented could be easily modified to tolerate approximately correct information. In an outdoor setting, the Geosensor can be implemented by a Global Positioning System (GPS) receiver. In an indoor environment, a Cricket [20] device may be a more effective Geosensor. In a static sensor network (for which GPS devices may be too expensive), synthetic coordinates (e.g., [17]) may be sufficient. We assume that each real mobile node receives an update from its Geosensor at least every t_{geo} units of time, where t_{geo} is a constant.

LocalCast Communication Service

The mobile nodes communicate using a local broadcast service, LocalCast, which is parameterized by a radius, R. When some node i performs a send$(m)_i$, the R-LocalCast service delivers the message — by a rcv$(m)_j$ — to every mobile node j within a radius R of the sender. We further assume that every message is delivered within d time, where d is a constant. The service has the following properties: (i) *Reliable Delivery*: Assume that the mobile node i performs a send$(m)_i$ action. Then for every mobile node j that is within distance R of the location of i when the message is sent, and remains within distance R of that location for d time and does not fail, a rcv$(m)_j$ event occurs within d time, delivering the message to node j. (ii) *Integrity*: For any LocalCast message m and mobile node i, if a rcv$(m)_i$ event occurs, then a send$(m)_j$ event precedes it, for some mobile node j. Intuitively, sending a message using this service should be thought of as making a single wireless broadcast (with a small number of retries, if necessary, to avoid collisions). We believe that for small R, this service is a reasonable (if simplistic) model of sending and receiving messages using wireless broadcast.

Formally, we assume that the LocalCast service has one or more sets of send/receive ports for each mobile node, and contains one or more message buffers. More specifically, we assume that the real nodes support an R_{RMN}-LocalCast service, for a constant R_{RMN}, which we call the RMN-LocalCast service. The RMN-LocalCast service has two sets of ports for each mobile node. The service contains two sets of message buffers, $messages[i]$ and $mpe\text{-}messages[i]$ for each $i \in I$, each of which temporarily hold messages destined for node i.

3 Virtual Mobile Nodes

The *VMN Abstraction* consists of both client mobile nodes and virtual mobile nodes (VMNs, also referred to as "virtual nodes"), which communicate using a LocalCast service. Throughout this paper, the term *mobile node* refers to any
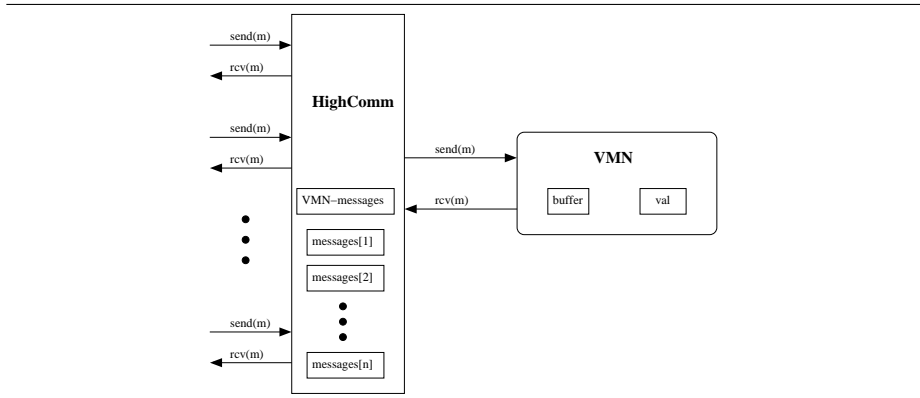
Fig. 1. Components of the VMN Abstraction. The VMN communicates with the clients using the VMN-LocalCast communication service.

node in the abstraction, a client or a VMN. Each mobile node is an arbitrary I/O automaton [16] (without tasks or fairness)[5]. A mobile node is prone to "crash-reboot" failures: a node may fail and recover. When a node recovers, it begins again in its initial state. A mobile node receives frequent updates from a Geosensor regarding the current time and its current location.

The key difference between clients and virtual nodes is that VMNs move in a predictable, predetermined path that is chosen in advance when the algorithm is specified. Clients, on the other hand, travel on an arbitrary path. Moreover, virtual nodes are robust. If the path of a VMN goes through a sparse region of the network, then the VMN fails during that interval of time; as soon as it reenters a dense region, it recovers.

For the rest of this paper, we assume that there is only a single VMN, communicating with several clients, as is depicted in Figure 1. Our results extend naturally to a model containing an arbitrary number of virtual nodes.

Formally, the VMN has two main state components: *VMN.val*, which represents the abstract state of the VMN I/O automaton, and *VMN.buffer*, a buffer that holds outgoing messages until they are ready to be sent.

Clients and virtual nodes communicate by sending messages using a Local-Cast service, as defined in Section 2[6]. Recall that the LocalCast service is parameterized by a radius, R. The VMN Abstraction implements an R_{VMN}-LocalCast service, for some fixed constant R_{VMN}, which we call VMN-LocalCast. We call the message buffers in the VMN-LocalCast service *messages*[i], for each $i \in I$, each of which holds messages destined for node i. If i is the identifier of the VMN, we refer to the *messages*[i] message buffer as *VMN-messages*.

---

[5] We expect that it is a simple extension to support timed and hybrid virtual nodes, instead of just I/O automata.

[6] Formally, we restrict the I/O automata to only two possible input actions: geo-update(t, ℓ) and rcv(m), and one possible output action: send(m).

Comm

send(m) · rcv(m)

messages[1]
messages[2]
⋮
messages[n]
ext−messages[1]
ext−messages[2]
⋮
ext−messages[n]

send(m) · rcv(m)

MPE val

send(m) · rcv(m)

MPE val

send(m) · rcv(m)

MPE val

lbcast(m) · lbcast−rcv(m)

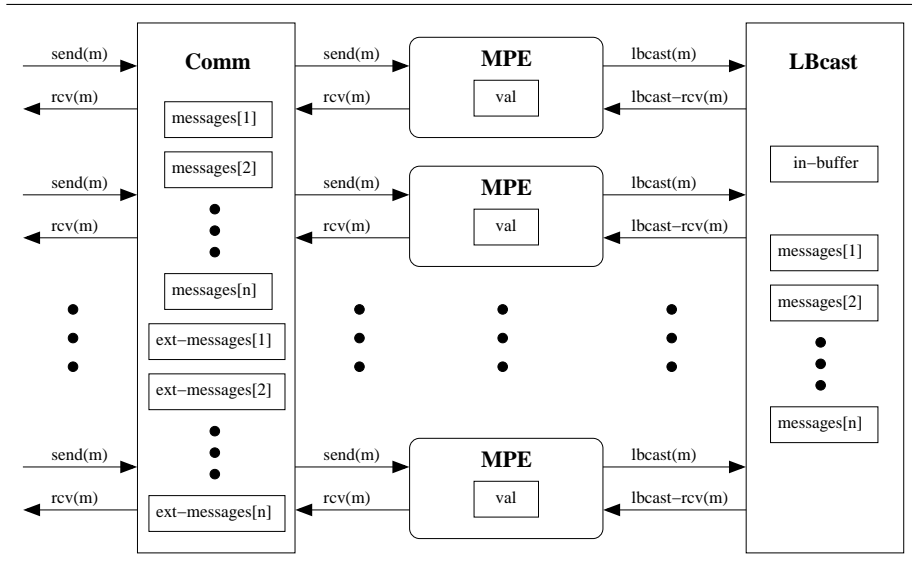**LBcast**

in−buffer

messages[1]
messages[2]
⋮
messages[n]

Fig. 2. Components of the VMN Abstraction implementation. The clients communicate with the *MPE* components using the RMN-LocalCast service; the *MPE* maintains consistent replicas using the TOBcast service.

4 Implementing the VMN Abstraction

In this section we present our implementation of the VMN abstraction. Recall that the VMN Abstraction consists of three components: virtual nodes, clients, and the VMN-LocalCast service. Our implementation consists of the Mobile Point Emulator (MPE) and the TOBcast service, which together implement the VMN and the VMN-LocalCast service. (The client automata, along with the automata to execute on the virtual node, are provided by the application developer, and hence no further discussion is necessary.) Formally, the MPE consists of one automaton, MPE_i, for every real node i. The relationship between these components is depicted in Figure 2.

Simple VMN Implementation

The simplest way to implement a VMN is by using a mobile "agent". An agent is a dynamic process that jumps from one real node to another, moving in the direction specified by the VMN path. An agent "hitches a ride" with a host that is near to the specified location of the VMN. This strategy has been used in the past to implement various services, such as group communication (see [8]). It can be generalized to support arbitrary I/O automata.

This simple algorithm meets one of the two goals of a VMN implementation: the movement of the virtual node is predictable. However, the host of the agent may fail, and therefore the VMN is not robust. For some applications, such as

simple routing, this may be sufficient. For many applications, however, this lack of robustness is undesirable. We use replication to solve this problem.

Mobile Points

We define a *mobile point* to be a circular region, of radius R_{mp}, that moves on the same path as the VMN: at time t, the center of the mobile point coincides with the preplanned location of the VMN at time t. (Even if the VMN has "failed", the mobile point—and the defunct VMN—conceptually continue along the pre-specified path.) Every real node that resides within a mobile point replicates the state of the virtual node.

Totally-Ordered Broadcast

Since the state of the VMN is replicated at multiple real nodes, the mobile point algorithm must maintain consistency among the replicas. We use the RMN-LocalCast communication service and the synchronized clocks to implement a totally-ordered broadcast service, which we call TOBcast, within the region defined by the mobile point.[7]

    We use a standard technique to implement the totally-ordered broadcast. A timestamp is affixed to each message, defining a total order. (Each node only sends a single message for each real time, and ties are broken using node identifiers.) Before delivering a message, the mobile node waits until at least time $d + 1$ has elapsed since the message was sent, ensuring that all earlier messages are received first. See Figure 3 for the pseudocode

---

TOBcast$_i$

Input TOBcast$(m)_i$, m a message
Input rcv$(m)_i$, m an internal message   2
Output TOBcast-rcv$(m)_i$, m a message
Output send$(m)_i$, m an internal message   4

State: 6
   clock, current real time
   location, current location   8
   incoming, initially $\emptyset$
   outgoing, initially $\emptyset$ 10

Input TOBcast$(m)_i$ 12
Effect:
   outgoing $\leftarrow$ *outgoing* $\cup$ 14
                  $\{\langle m,\ i,\ clock,\ location\rangle\}$
   16

Output send$(m)_i$
Precondition: 18
   $m \in outgoing$
Effect: 20
   outgoing $\leftarrow$ *outgoing* $/\ \{m\}$
   22

Input rcv$(\langle m,\ j,\ t,\ \ell\ \rangle)_i$
Effect: 24
   incoming $\leftarrow$ *incoming* $\cup\ \{\langle m,\ j,\ t,\ \ell\ \rangle\}$
   26

Output TOBcast-rcv$(m)_i$
Precondition: 28
   $\langle m,\ j,\ t,\ \ell\ \rangle \in incoming$
   $t+d+1 = clock$ 30
   $\nexists\ \langle m',\ j',\ t',\ *\rangle \in incoming$ s.t. $j' < j$
Effect: 32
   incoming $\leftarrow$ *incoming* $\setminus\ \{\langle m,\ j,\ t,\ \ell\ \rangle\}$
   34

Trajectories:
satisfies   36
   $d(time) = 1$
   $d(location[i])$ is arbitrary   38
   constant *incoming*, *outgoing*
stops when   40
   $\exists\ \langle m,\ j,\ t,\ \ell\ \rangle \in incoming$ **such that**:
     $t+d+1 = time$ 42
   $\exists\ \langle m,\ j,\ t,\ \ell\ \rangle \in outgoing$

Fig. 3. The TOBcast service implementation.

---

[7] The TOBcast service takes the place of the "LBcast" service used in [6].

Fig. 4. MPE State for Node i and VMN h for IOA $\tau = \langle sig, states, start, \delta \rangle$

(using the TIOA formalism [11]) that implements the totally-ordered broadcast service, which we call TOBcast.

Theorem 1. *The TOBcast service guarantees that messages are delivered in the order in which they are sent (according to real time), and if a real node within a mobile point sends a message, then every other real node in the mobile point (that resides in the mobile point for the duration of the broadcast) receives the message.*

The Mobile Point Emulator

The Mobile Point Emulator is based on a replicated state machine technique similar to that originally presented in [14], augmented to support joins, leaves, and recovery. The MPE replicates the state of the VMN at every node within the mobile point's region. It uses the total ordering of messages to ensure that the replicas are updated consistently. The state of each MPE_i is given in Figure 4, and the signature of each MPE_i is given in Figure 5. The algorithm itself is in Figure 4. (All the line numbers in this section refer to Figure 4.)

MPE State. The *status* of the Mobile Point Emulator at node i transitions between four *status* value: idle, indicating that the real node is not within the mobile point, joining or listening, indicating that the real node is in the process of joining the VMN, or active, indicating that the real node is participating in the VMN emulation.

When a node is active in the mobile point, it maintains a replicated copy of the state of the virtual node, *val*. The MPE maintains a queue of *pending-actions*, which are processed in order. The TOBcast service is used to ensure that each MPE processes the pending actions in the same order. The *completed-actions* and *answered-joins* store information on which actions have already been processed, thus preventing a message from being processed twice. The *join-id* is used during the join protocol.

The *mp-location* variable maintains the current location of the center of the mobile point. This may be continually changing, however it is a predetermined function of time. The *location* variable maintains the current location of the real node on which the MPE is executnig. The *clock* maintains the current real time.

MPE Transitions. The MPE_i modifies the replicated state, *val* only when the real node receives a TOBcast message indicating that a particular action should be performed. Since all active nodes process the TOBcast messages in the same order, all nodes modify their state in the same way, thus maintaining consistent replicas.

When an active node, i, receives a message destined for the VMN—that is, a rcv$_i$ occurs—it immediately resends it to the other replicas using the TOBcast service (lines 1–4). When the TOBcast service delivers the message, each node modifies its replica, performing a VMN rcv of that message (lines 65–69 and lines 48–61).

Sometimes, a VMN chooses to initiate an internal or output action. In this case, an active node determines that a certain action is enabled, and broadcasts a message to the other replicas (lines 12–20). As in the previous case, when the TOBcast service delivers the message, each node modifies the state of its replica, performing the specified VMN action (again, lines 65–69 followed by lines 48–61). In some cases, this causes the VMN itself to send a message (line 61).

Joining a Mobile Point. Whenever a real node is within the perimeter defined by the mobile point, it initiates the join protocol (lines 22–30); whenever a real node is outside of a mobile point, it executes the leave protocol, which reinitializes its states and sets its status to idle (lines 42–46). The maximum speed of the VMN is effectively determined by the speed of the join protocol and the speed of the real nodes: the mobile point must move slowly enough so that new nodes can enter and join the mobile point before the old nodes leave.

The join protocol for node i begins when i broadcasts a join-req, requesting a copy of the current state (line 30). When node i receives the TOBcast for its own join request, it enters the listening state (lines 71–72). This indicates that node i can begin to monitor the messages in the system. In particular, it saves any messages that it cannot yet process in *pending-actions*.

When some active node, j, receives a join request, it sends a join acknowledgment, join-ack (lines 73–78). This acknowledgment includes a copy of its replica
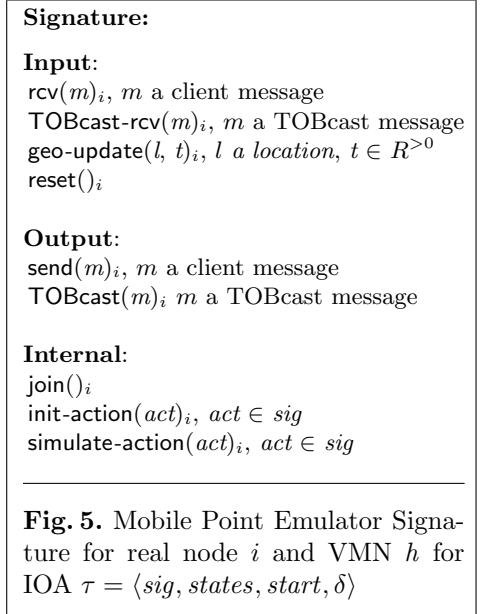
of the virtual node, val_j. When i receives the join acknowledgment, it copies the replicated state (lines 79–84), and begins to process its *pending-actions*.

Recovery. The Mobile Point Emulator simulates VMNs that are quite robust: they fail only when they enter a depopulated region of the network. However, as soon as all the nodes leave a mobile point, the virtual node loses its state. The Mobile Point Emulator contains a recovery mechanism that restarts the virtual node in this case. When a real node enters the mobile point and cannot communicate with any other active nodes, it can choose to broadcast a reset message (lines 32–35). (Should it choose not to, the VMN may not recover.) When a node receives a reset message it reinitializes its state (lines 85–87). In particular, when the node that discovers the mobile point has failed receives its own reset message, it restarts the mobile point.

VMN-LocalCast

When a client sends a message to the VMN using the VMN-LocalCast service, three steps occur: first, the client uses the RMN-LocalCast service to send the message to a real node in the VMN; second, the real node in the VMN rebroadcasts the message using the TOBcast service; finally, each node in the mobile point processes the message, and the VMN receives the message. Therefore, if the underlying real nodes deliver messages within time d, then the VMN-LocalCast guarantees that messages are delivered within time $2d+1$: it takes time d for the real node to receive the message from the client, and an additional time $d+1$ for the TOBcast service to redeliver the message.

The same process occurs (partially in reverse) when the VMN sends a message to a client: first a real node in the VMN broadcasts the intent of the VMN to send a message using the TOBcast service; second, the real nodes in the VMN process the message (at which point the VMN has buffered the outgoing message); third, some real node uses the RMN-LocalCast service to send the message to the client.

Recall that the VMN-LocalCast service has a range of R_{VMN} and the RMN-LocalCast service has a range of R_{RMN}. In order for the algorithm to be correct we assume that:
$$R \geq 2R_{VMN} + 2t_{geo} \cdot v_{max} \ .$$

There are two reasons why the extra broadcast range is necessary. First, a real node that is at distance R_{VMN} from the center must be able to send a message to any client that is at distance R_{VMN} from (the center of) the VMN; thus a radius of $2R_{VMN}$ is necessary. Second, a real node only receives updates about its location every t_{geo} time units; therefore, a real node may be an additional $t_{geo} \cdot v_{max}$ distance outside the mobile point before detecting that it is no longer a part of the VMN emulation.

VMN Performance

Each step of the VMN automata requires at most one TOBcast message to be sent, which takes time $d+1$; no other delay is incurred. Therefore, the Mobile

Input rcv$(m)_i$
2   **Effect:**
    $TOBcast\text{-}out \leftarrow TOBcast\text{-}out\ \cup$
4        $\{\langle\textsf{simulate}, \langle\textsf{rcv}, m\rangle, \bot\rangle\}$

6 **Output** send$(m)_{h,i}$
  Precondition:
8     $m \in local\text{-}out$
  Effect:
10     $local\text{-}out \leftarrow local\text{-}out\ /\ \{m\}$

12 **Internal** init-action$(act)_{h,i}$
  Precondition:
14     $status = \textsf{active}$
    $|mp\text{-}location - location| < R$
16     $\delta(val,\ act) \neq \bot$
  Effect:
18     $temp\text{-}oid \leftarrow \langle clock, i\rangle$
    $TOBcast\text{-}out \leftarrow TOBcast\text{-}out\ \cup$
20       $\{\langle\textsf{simulate}, act, temp\text{-}oid\rangle\}$

22 **Internal** join$()_i$
  Precondition:
24     $|mp\text{-}location - location| < R$
    $status = \textsf{idle}$
26   **Effect:**
    $join\text{-}id \leftarrow \langle clock, i\rangle$
28     $status \leftarrow \textsf{joining}$
    $TOBcast\text{-}out \leftarrow TOBcast\text{-}out\ \cup$
30       $\{\langle\textsf{join-req}, \bot, join\text{-}id\rangle\}$

32 **Input** reset$()_i$
  Effect:
34     $TOBcast\text{-}out \leftarrow TOBcast\text{-}out\ \cup$
        $\{\langle\textsf{reset}\rangle\}$
36 **Input** geo-update$(\ell,\ t)_i$
  Effect:
38     $location \leftarrow \ell$
    $clock \leftarrow t$
40     $val \leftarrow \delta(val,$
      $\langle\textsf{geo-update}, t, mp\text{-}location\rangle)$
42   **if** $(|mp\text{-}location - location| \geq R)$
      and $(status \neq \textsf{idle})$ **then**
44     $status \leftarrow \textsf{idle}$
    $val \leftarrow start(\tau)$
46     $pending\text{-}actions \leftarrow \emptyset$

Internal simulate-action$(act)_i$ 48
Precondition:
  $status = \textsf{active}$ 50
  $|mp\text{-}location - location| < R$
  $head(pending\text{-}actions) = \langle\textsf{simulate}, act, oid\rangle$ 52
Effect:
  Dequeue$(pending\text{-}actions)$ 54
  if $(\langle\textsf{simulate}, act, oid\rangle \in completed\text{-}actions)$
    then continue; 56
  if $(\delta(val,\ act) = \bot)$ **then continue**;
  $val \leftarrow \delta(val,\ act)$ 58
  $completed\text{-}actions \leftarrow completed\text{-}actions\ \cup$
      $\{\langle\textsf{simulate}, act, oid\rangle\}$ 60
  if $(act = \langle\textsf{send}, m\rangle)$ **then** send(m)
 62
Input TOBcast-rcv$(\langle optype, param, oid\rangle)_i$
Effect: 64
  if $(optype = \textsf{simulate})$ **then**
    if $(status \neq \textsf{listening}\ \textbf{or}\ \textsf{active})$ **then** 66
      continue;
    else Enqueue$(pending\text{-}actions,$ 68
        $\langle\textsf{simulate}, param, oid\rangle)$
  else if $(optype = \textsf{join-req})$ **then** 70
    if $((status = \textsf{joining})\ \textbf{and}\ (oid = join\text{-}id))$
      then $status \leftarrow \textsf{listening}$ 72
    if $((status = \textsf{active})$ **then**
      if $(oid \in answered\text{-}joins))$ **then** 74
        continue;
      else if $(|mp\text{-}location - location| < R)$ 76
        then TOBcast$(\langle\textsf{join-ack},$
          $\langle val, completed\text{-}actions\rangle, oid\rangle)$ 78
  else if $(optype = \textsf{join-ack})$
    $answered\text{-}joins \leftarrow answered\text{-}joins\ \cup\ \{oid\}$ 80
    if $(status = \textsf{listening})$ **then**
      if $(oid = join\text{-}id))$ **then** 82
        $status \leftarrow \textsf{active}$
        $\langle val, completed\text{-}actions\rangle \leftarrow param$ 84
  else if $(optype = \textsf{reset})$ **then**
    $status \leftarrow \textsf{active}$ 86
    $pending\text{-}actions \leftarrow \emptyset$
 88
Output TOBcast$(m)_i$
Precondition: 90
  $m \in TOBcast\text{-}out$
Effect: 92
  $TOBcast\text{-}out \leftarrow TOBcast\text{-}out\ /\ \{m\}$

Fig. 6. Automaton $MPE_{h,i}$ running on node i implementing the VMN executing IOA $\tau = \langle sig, states, start, \delta\rangle$

Point Emulator ensures that a program executing on a VMN is slowed by at most a factor of $d + 1$.

Theorem 2. *The Mobile Point Emulator and the TOBcast service (and the trivial client implementation) correctly implement the VMN Abstraction. More formally: let A be the abstract VMN model, and let S be the implementation. Then timed-traces(S) ⊆ timed-traces(A)[8].*

5 Correctness of the Mobile Point Emulator

In this section, we present a sketch of the proof that the Mobile Point Emulator correctly implements the VMN abstraction. We demonstrate a forward simulation relation [11] between the implementation described in Section 4 and the VMN Abstraction described in Section 3, which implies the correctness of our algorithm. For more details, see [5].

The simulation relation consists of five main conditions. The first two conditions relate messages in the RMN-LocalCast service and messages in the VMN-LocalCast service. Condition 1 relates incoming messages: if m is a message in *RMN-LocalCast.mpe-messages*[i] waiting to be delivered to some mobile node i, then message m is also waiting in *VMN-LocalCast.VMN-messages* to be delivered to the VMN. Condition 2 relates outgoing messages: if m is a message in *RMN-LocalCast.messages*[i] being sent by some Mobile Point automaton to node i, then message m is also in *VMN-LocalCast.messages*[i].

Condition 3 relates the replicated state of a Mobile Point Emulator to the state of the abstract VMN: for all active mobile nodes i that have completed the join protocol, if you start with the state represented by *MPE.val$_i$*, and process all the pending actions in *MPE.pending-actions$_i$* in the order they are stored in the queue and all the messages waiting in the TOBcast queue *TOBcast.outgoing*[i] (again, in the order they are stored in the queue), then the resulting value is equivalent to the state of the VMN, *VMN.val*.

Condition 4 is used to show that the join protocol works: if v is a state contained in a join acknowledgment message stored anywhere in the system, then if you start with the state represented by v and process all the messages in the *MPE.pending-actions$_i$* queue that are sent after the associated join request, then the resulting value is equal to the state of the VMN, *VMN.val*.

Condition 5 ensures that if the implementation initiates a send, then the VMN can perform a send: if m is a message indicating that a send(x) is to occur, and m is either in a TOBcast queue (*TOBcast.messages*[i]) or waiting to be performed (in *MPE.pending-actions$_i$*), then the message x is in *VMN.buffer*.

We claim that the Mobile Point Emulator correctly implements the VMN abstraction, in that any service built on the VMN abstraction runs correctly on the Mobile Point Emulator:

---

[8] The *timed-traces* of a system capture the externally visible behavior and the times at which the external visible events occur.

Proof of Theorem 2 (sketch). In the initial state, the five conditions described above hold: all the message queues and buffers are empty, and the $MPE.val$ component of the replicated automata is set to the initial state, as is the $VMN.val$ component.

We proceed by induction, examining all possible actions in the implementation, and determining a suitable sequence of actions in the abstract model. For example, assume that a client attempts to send a message to the VMN. In the implementation, this results in a message being added to the communication service's message queues; in the abstract model, this results in a message being added to the high-level communication service's VMN message queue, preserving Condition 1.

One interesting case occurs when a mobile node broadcasts a message using the TOBcast service indicating that a transition of the VMN automaton should occur. In the low-level implementation, a message is added to $TOBcast.messages[i]$, for all nodes i. In order to maintain Condition 5, it is necessary to immediately perform the required transition in the VMN, updating $VMN.val$. If the required transition is an output action, the VMN sends a message, placing it in $VMN.buffer$, thus maintaining Condition 5. We omit the many remaining cases.

The conditions are also maintained when time passes: if a node is far enough away that it does not receive a LocalCast message, then it has left the focal point. We conclude that Conditions 1–5 are a forward simulation relation. □

6 Algorithms for Virtual Mobile Nodes

To demonstrate the utility of the new approach, we briefly discuss several basic algorithms that use VMNs to solve interesting problems simply and efficiently. For more details, see [5].

Consider the problem of routing messages. The simplest algorithm to route message relies on a single virtual node traversing the network collecting messages and delivering them. It is possible to adapt the compulsory protocols of Chatzigiannakis et al. [4], yielding alternate message delivery services that can operate in a non-compulsory framework. Routing a message to a virtual node is even simpler: the current location of a virtual node is known in advance, so we can route messages directly to the predicted location of the virtual node.

Virtual nodes can also be used to collect sensor data, traversing the network. Instead of maintaining a complicated dynamic data structure of sensor readings, a virtual node can aggregate data as it is collected and process complex queries.

Finally, we suggest that VMNs may be useful for a number of common generic services. Group communication services (e.g., as in [9, 12, 13]) can be implemented by adapting the strategy in [8] to use a robust virtual node, instead of a fragile token, to collect and deliver group information. An atomic memory service can be constructed using the approach developed in [6]; in this case, however, the data can be programmed to travel around the network.

7 Discussion and Concluding Remarks

We have presented a new technique for implementing algorithms in mobile *ad hoc* networks. In general, it is difficult to devise algorithms for such chaotic, unpredictable environments. The VMN abstraction makes the task easier by providing robust virtual nodes that move in a predictable manner. Moreover, we have presented the Mobile Point Emulator, a new algorithm that allows real mobile nodes to emulate reliable virtual nodes, using location information and a basic (though powerful) local communication service.

We believe that the VMN abstraction and low-level algorithms similar to the Mobile Point Emulator can significantly simplify the development of application-level algorithms for mobile networks.

There are a number of limitations, however, to the Mobile Point Emulator. It depends on a powerful local communication service, and the correctness of the algorithm depends on both the reliability and timeliness of the service. Moreover, it assumes that the algorithm is executing in a trusted environment; it remains an open question to consider the security implications, and whether such a solution could work in a more hostile environment. Finally, the Mobile Point Emulator is an expensive algorithm, requiring significant amounts of communication and power consumption.

Engineering and Experimentation. There are many ways in which the Mobile Point Emulator can be optimized for implementation purposes. For example, if a (temporary) leader is elected within a mobile point, and the leader initiates all the transitions for the replica, conflicting requests are avoided and power is saved. As a second example, when a node leaves a mobile point, it need not wholly reset its replica state; on rejoining the mobile point, the join acknowledgment only needs to contain the changes in the state. It would be interesting to experiment with a real implementation of VMNs to determine the extent to which the algorithms can be optimized, and whether the utility outweighs the implementation overhead.

Self-Stabilizing VMNs. Long-term robustness of the VMN abstraction could be improved if the virtual nodes could tolerate transient faults, such as state-corruption or a violation of the broadcast assumptions. It is an interesting open question whether the Mobile Point Emulator can be made self-stabilizing.

Dynamic Virtual Mobile Nodes. We have assumed that the set of VMNs and their paths are fixed in advance. For some applications, this is sufficient; however, for others, it would be useful if the paths of the VMNs could be determined on-the-fly. For example, one can imagine using a VMN to follow a moving entity, either performing a service for that entity, or tracking the location of the entity. Dynamic paths can also be used to help VMNs avoid unpopulated areas of the mobile network, thus improving robustness. It may also be useful to generate virtual nodes dynamically; for example, a new VMN might be generated to track every entity that enters a certain geographical area.

References

1. J. Beal. Persistent nodes for reliable memory in geographically local networks. Tech Report AIM-2003-11, MIT, 2003.
2. J. Beal. A robust amorphous hierarchy from persistent nodes. In *Proc. of Communication Systems and Networks*, 2003.
3. T. Camp, Y. Liu. An adaptive mesh-based protocol for geocast routing. *Journal of Parallel and Distributed Computing: Special Issue on Mobile Ad-hoc Networking and Computing*, pp. 196–213, 2002.
4. I. Chatzigiannakis, S. Nikoletseas, P. Spirakis. An efficient communication strategy for ad-hoc mobile networks. In *Proc. 15th International Symp. on Distributed Computing*, 2001.
5. S. Dolev, S. Gilbert, N. A. Lynch, E. Schiller, A. A. Shvartsman, J. L. Welch. Virtual mobile nodes for mobile adhoc networks. Tech Report LCS-TR-937, MIT, 2004.
6. S. Dolev, S. Gilbert, N. A. Lynch, A. A. Shvartsman, J. Welch. Geoquorums: Implementing atomic memory in mobile ad hoc networks. In *Proceeding of the 17th International Conference on Distributed Computing*, 2003.
7. S. Dolev, S. Gilbert, N. A. Lynch, A. A. Shvartsman, J. L. Welch. Geoquorums: Implementing atomic memory in ad hoc networks. Tech Report LCS-TR-900, MIT, 2003.
8. S. Dolev, E. Schiller, J. Welch. Random walk for self-stabilizing group communication in ad-hoc networks. In *Proc. of the 21st IEEE Symp. on Reliable Distributed Systems*, 2002.
9. *Communications of the ACM, Special section on Group Communication Systems*, volume 39(4), 1996.
10. K. P. Hatzis, G. P. Pentaris, P. G. Spirakis, V. T. Tampakas, R. B. Tan. Fundamental control algorithms in mobile networks. In *Proc. of the 1st ACM Symp. on Parallel Algorithms and Architectures archive*, Saint Malo, France, 1999.
11. D. K. Kaynar, N. Lynch, R. Segala, F. Vaandrager. The theory of timed I/O automata. Tech Report MIT-LCS-TR-917a, MIT, 2004.
12. I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, Hebrew University, Jerusalem, 1994. URL: http://www.cs.huji.ac.il/simtransis/publications.html.
13. I. Keidar, D. Dolev. Efficient message ordering in dynamic networks. In *Proc. of the 15th annual ACM Symp. on Principles of distributed computing*, 1996.
14. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
15. Q. Li, D. Rus. Sending messages to mobile users in disconnected ad-hoc wireless networks. In *Proc. of the 6th MobiCom*, 2000.
16. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.
17. R. Nagpal, H. Shrobe, J. Bachrach. Organizing a global coordinate system from local information on an ad hoc sensor network. In *2nd Workshop on Information Processing in Sensor Networks*, 2003.
18. B. Nath, D. Niculescu. Routing on a curve. *ACM SIGCOMM Computer Communication Review*, 33(1):150 – 160, 2003.
19. J. C. Navas, T. Imielinski. Geocast – geographic addressing and routing. In *Proc. of the 3rd MobiCom*, 1997.
20. N. B. Priyantha, A. Chakraborty, H. Balakrishnan. The cricket location-support system. In *Proc. of the 6th MobiCom*, 2000.