The Virtual Node Layer: A Programming Abstraction for Wireless Sensor Networks*

Matthew Brown
mdbrown@mit.edu

Seth Gilbert
sethg@mit.edu

Nancy Lynch
lynch@theory.csail.mit.edu

Calvin Newport
cnewport@mit.edu

Tina Nolte
tnolte@mit.edu

Michael Spindel
mspindel@mit.edu

MIT Computer Science and Artificial Intelligence Lab
Cambridge, MA 02139, USA

ABSTRACT

The Virtual Node Layer (VNLayer) programming abstraction provides programmable, predictable automata—virtual nodes—emulated by the low-level network nodes. This simplifies the design and rigorous analysis of applications for the wireless sensor network setting, as the layer can mask much of the uncertainty of the underlying components. In this paper, we define a general VNLayer architecture, and then use this framework to design a practical VNLayer implementation, optimized for real-world use. We then discuss our experience deploying this implementation on a testbed of hand-held computers, and in a custom-built packet-level simulator, and present a sample application—a virtual traffic light—to highlight the power and utility of our abstraction. We conclude with a survey of additional applications that are well-suited to this setting.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Theory, Algorithms, Reliability

Keywords

Wireless ad hoc networks, network architecture, virtual infrastructure

*This work is supported by USAF,AFRL Award #FA9550-04-1-0121, NSF award #CCR-0121277, Research Foundation of CUNY Subcontract 71081-00-01, Cisco Systems, and Quanta-MIT Award Number 012627-009.

1. INTRODUCTION

The increased miniaturization of computing devices and radios presents new opportunities for the design and implementation of low-cost network systems. One can imagine, for example, a network of small, inexpensive, radio-equipped computers being used to monitor environmental conditions, coordinate robots, or manage traffic on highways or in the air. Such platforms, however, are generally not well-behaved. They are *ad hoc*, in that they include little (or no) fixed network infrastructure. In addition, the set of participating network nodes is typically not known in advance, making it necessary to self-configure the nodes into coherent structures. Furthermore, these networks are *dynamic*, in that the set of nodes can change over time, as devices enter and leave the area of interest, fail and recover. Finally, we must acknowledge the prevalence of message loss due to wireless interference or noise.

As a result, it is difficult to write and debug applications that will run predictably on these fundamentally unpredictable platforms. To obtain predictable behavior, the developer must grapple with the whole range of complexities described above. Even implementations of simple applications can become unwieldy under the dynamic conditions that define many wireless ad hoc network deployments.

A promising solution to this problem is clean, well-defined *abstraction layers*—high-level, well-behaved network models emulatable by the low-level network nodes. These layers can mask much of the uncertainty and change inherent in this setting, simplifying the design of applications. They can also simplify *rigorous analysis*. Such analysis is crucial for command and control contexts—such as using actuator-equipped sensor nodes to control a factory production line—where best-effort is not sufficient, and applications must behave within precise specifications. This approach, in essence, concentrates the challenge of handling the unpredictable nature of the network in the (one-time) task of carefully designing and analyzing the abstraction layer and its implementation.

With this in mind, it is clear that the careful design of good abstraction layers for wireless ad hoc networks should be a major research endeavor. Though other researchers have considered the use of abstraction layers for sensor networks

(c.f. [4, 15]), much of this prior work has focused on simplifying the design of best-effort applications, such as data aggregation and analysis. Cluster-based networks (e.g., [14]) represent another useful tool to aid fault-tolerant data aggregation and efficient routing. They do not provide, however, a fully-generalized programming abstraction.

1.1 The Virtual Node Layer

The *Virtual Node Layer (VNLayer)* is a programming abstraction that presents the application developer with two types of entities to program: predictable *Virtual Nodes (VNs)*, and unpredictable *Client Nodes (CNs)*, which correspond to the physical nodes in the system. See, for example, the VNLayer described in Figure 1. The black circles represent physical nodes (and their corresponding CNs), and the white rectangles represent the VNs.

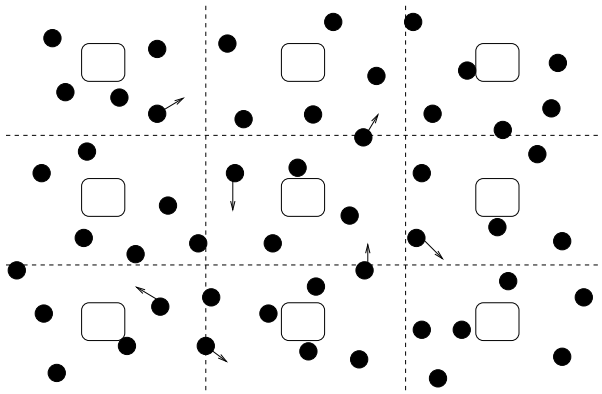

Figure 1: In this VNLayer, the mobile, fault-prone physical nodes (black circles) emulate stable Virtual Nodes (white rectangles) found in known grid regions. The arrows emphasize that physical nodes can be mobile while the Virtual Nodes remain stationary.

CNs share the dynamic nature of their physical counterparts: they can fail unpredictably, and the set of CNs in a given region is unknown a priori and can change over time.

VNs, on the other hand, are not intended to correspond directly to the underlying physical network. Rather, the VNs are identified with arbitrary regions of the network, and may either remain in a fixed, known location or move in a controlled manner through the network. In the example of Figure 1, each VN is at a fixed location, whose region corresponds to the surrounding grid square.

Each VN is emulated by the physical nodes in its region, using distributed-algorithms based on replicated state machines, elected leaders, and quorums. This emulation can be performed in a fault-tolerant manner, allowing the VNs to avoid much of the dynamism of the underlying network.

Consider how this VNLayer could simplify the task of distributed agreement in a given region. Instead of writing a distributed algorithm that must take into account the fact that the set of participating physical nodes is unknown, and those participating might fail or leave during the protocol, the developer could deploy a simple distributed agreement object implementation on the stable VN in the region. Following the standard approach, the CNs can propose values to their local VN, which will remember the first proposal, and return this value to all who request the decision. As desired, the complexity of dealing with the dynamic nature of the participants has been concentrated into the design and analysis of the VNLayer.

1.2 Prior Work

Our research group has been working on the design and analysis of several varieties of VNLayers and their implementations [8, 6, 7, 9, 13]. We have used rigorous formal modeling tools, such as the Timed I/O Automata formalism, to analyze the correctness of these implementations. We have also designed and analyzed a growing collection of applications to run on top of these layers. These include, among others, geocast communication, robot coordination, and object tracking solutions.

These existing VNLayers can be divided into two main categories. Implementations in the first category make optimistic assumptions about the communication behavior of the physical nodes in the network. Namely, they assume that broadcast messages will be always be received by nodes within range (perhaps with some bounded delay). Under these conditions, in [8], we first define a VNLayer whose VNs are stationary asynchronous objects. In [6], we present a VNLayer in which the VNs are arbitrary timed asynchronous automata, positioned at known fixed locations, as pictured in Figure 1. The failure model allows crash failures if and only if the region immediately surrounding the VN is unpopulated by physical nodes. It also allows some bounded stretch in the clock rate (necessitated by delays caused by the operations of the underlying emulation).

In the VNLayer of [8], the VNs are untimed asynchronous automata moving along pre-defined trajectories, known to all CNs. The failure model specifies that VNs can suffer crash failures if and only if their path takes them into a region unpopulated by any physical nodes. A variant of this layer, [9], has the VNs re-calculate their path in real time. This allows them, for example, to avoid unpopulated regions, reducing the possibility of VN crashes.

The second category of VNLayer implementations makes more pessimistic assumptions about the communication behavior of physical nodes, allowing for arbitrary, non-uniform message loss. It does assume, however, rough synchrony (as obtained, for example, by approximate clock synchronization, which is a well studied problem). In an on-going project, we describe a VNLayer in which the VNs are arbitrary synchronous automata positioned at known fixed locations; fault-tolerant agreement protocols, such as those explored in [5], are used to maintain agreement on VN state in the face of message loss.

1.3 The Contributions of this Study

In this paper we present a novel VNLayer optimized for practical use. We start, in Section 2, by describing a general architecture for VNLayer design. In Section 3, we provide a high-level description of our new VNLayer and its implementation. To verify its practicality, we then describe our

experience deploying this abstraction on a testbed of handheld computers and within a custom-built packet-level simulator. In Section 4, we demonstrate its utility by describing a demo application: a virtual traffic light. We conclude, in Section 5, with examples of other applications whose implementations could benefit from the use of VNLayers.

2. THE VNLAYER ARCHITECTURE

To implement the VNLayer abstraction, the physical nodes run emulation software that maintains a consistent view of the layer. One way to do this, for example, is to have all the physical nodes in a given region run that region's VN application, locally transforming their local views of the state synchronously. Another—albeit non-fault-tolerant—approach would be for a single leader per region to run the VN application, passing off control to a new leader if it leaves the region.

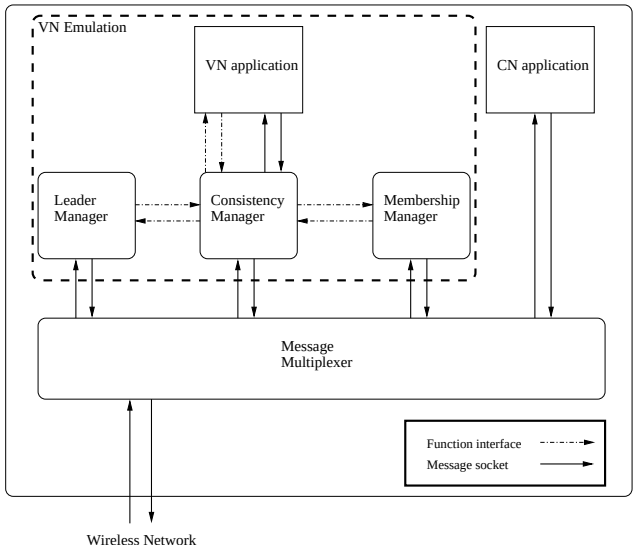

Figure 2: VNLayer architecture at each physical node.

We desire an emulation architecture that remains general enough to facilitate the design and deployment of a wide variety of emulation schemes. At the same time, however, we want it to be sufficiently modular that practitioners can avoid producing complicated, monolithic emulator implementations for each new network context.

2.1 Architecture Overview

To meet these goals, we propose the architecture outlined by Figure 2. In this framework, the task of emulating a particular VN application program is divided among three main components, each of which has access to location information accurate to region granularity:

Leader Manager: This component attempts to elect a region leader in each populated region of the network. It implements a boolean function, called exclusively by the Consistency Manager, which returns the leader status of the physical node on which it is running.

Membership Manager: This component retrieves the current state of the VN application being emulated in the region. It implements a function, called exclusively by the Consistency Manager, which triggers this retrieval and returns either the state or a time-out (e.g., the latter may occur if there are no other nodes in the region).

Consistency Manager: This component contains the main logic of the emulation. It is ultimately responsible for keeping the VN application state synchronized with the other physical nodes in the region. To do so, it can call upon both the leader and membership manager, and can send out its own control traffic (e.g., perhaps as part of a distributed agreement protocol).

Access to the VN application is mediated through the Consistency Manager, which implements the incoming and outgoing message sockets used in the application—passing it messages and receiving/processing its outgoing messages. We require that all VN applications implement two interface functions called exclusively by the Consistency Manager: *getState* and *updateState*. The former returns the VN application's current state and the latter resets the VN application's state to the values passed as a parameter to the function. This interface allows the Consistency Manager to keep the emulated application synchronized with other nodes in the region. For example, if a physical node enters a new region it might need to switch the application state to that of the application being emulated in the new region.

The CN application(s) are unrestricted and run outside of the emulation apparatus, sending and receiving messages to and from the network directly.

3. THE "REACTIVE VN" VNLAYER

Here we describe a new VNLayer implementation. This layer provides programmable *Reactive VNs*, which are receive event-driven automata. That is, their operations are defined exclusively in terms of a *msgReceived* handler which, upon being called with a received message, can transform the automata state and (potentially) return message(s) to be broadcast in response. The VNs are found at fixed, known locations in the network, and fail if and only if their region contains no physical nodes. We assume that messages might be lost due to collisions.

We choose the Reactive VN paradigm for this VNLayer as it simplifies the task of coding applications. That is, it is easier to code a message handler than it is, for example, to code an arbitrary timed I/O automaton. Notice that these VNs do not have direct access to timers. It turns out, however, that in situations where time is needed (e.g., the VN waits for a certain timeout before broadcasting), it is often sufficient to have the CNs (who do have access to clocks) broadcast regular updates of the current time.

In the following, we provide a high-level description of the main components in our VNLayer imlementation. We then describe our experience deploying the layer on a testbed of HP iPAQ Pocket PC's and in a custom-built packet-level simulator. We conclude with a summary of future work.

3.1 Implementation Details

VN application: Each VN application is receive event-driven. That is, the application is written as a *msgReceived* function, which is passed an incoming message. This function can arbitrarily transform the application state, and then

returns the message(s), if any, that it wishes to broadcast. An example of a *msgReceived* function can be found in Section 4.

Leader Manager: Our implementation of this component relies on a pulse-based algorithm. A leader sends out a pulse at regular intervals. If the leader's pulse times out, the algorithm attempts to declare its physical node as leader. If multiple nodes attempt to declare themselves leader, they elect the one with the lowest ID. If multiple nodes elect themselves leader—due, perhaps, to message loss at the point of declaration—they will eventually notice this situation by hearing each other's pulses. A leader who hears a pulse from another leader with a lower ID will relinquish his status.

Membership Manager: A simple join protocol asks the leader for a serialized version of the emulated VN application's state. If no leader exists, the request times out.

Consistency Manager: In our implementation, *all* physical nodes in a region locally emulate the VN application. Only the current leader, however, broadcasts on behalf of the application. For example, assume a client sends a message to the VN in a given region. All physical nodes in the region receive this message. The Consistency Manager running on each of these nodes passes the message to the VN application, triggering, perhaps, a state change. If the VN application reacts by broadcasting a message, the Consistency Manager checks to see if its node is the leader. If it is, it passes the broadcast messages out onto the network. Otherwise, it discards it.

We also have the Consistency Manager perform several other crucial functions. For example, it executes a message ordering algorithm on all received application messages, before they are passed to the local copy of the VN application. The algorithm maintains a consistent total order on the messages across all nodes in the region.[1]

To account for the possibility that a node might miss a message due to collision or temporary obstruction (possibly leading the local copy of the emulated VN application to fall out of synch with the copies on other nodes in the region), we have the Consistency Manager running on the current leader tag each outgoing VN application message with a hash of the application's state. When a non-leader node receives this message, its Consistency Manager can check that the hash tag matches the hash of its local copy of the VN application state. If the match fails, the node is out of synch, and it refreshes its state by having the Membership Manager perform a join protocol.

Finally, the Consistency Manager is also responsible for triggering the Membership Manager to initiate a join protocol whenever the physical node has entered a new region.

[1]This was necessary as our testing revealed that 802.11 broadcast does not guarantee total ordering. For example, imagine two nodes, A and B, which broadcast (near) simultaneously. The nodes both receive their own messages instantly, as the loop-back functionality of the network layer bounces a copy of the message into the receive buffer before it attempts the actual broadcast. Next, after back-off related delays, both messages are broadcast and received by the other receiver. At this point, A's buffer is ordered M_A, M_B, while B's buffer is ordered M_B, M_A.

3.2 Testbed Deployment

To obtain platform independance, we coded our implementation using Python, an interpreted object-oriented language which runs on most Windows, Mac, and Linux-based operating systems. Our implementation was intended for mobile devices, such as palm or laptop computers, communicating wirelessly using the 802.11 standard.

We verified the system through a sample deployment on a testbed comprising of five HP iPAQ Pocket PCs, running linux, and communicating with Netgear Compact Flash 802.11b Wireless Adaptors set to the same frequency in ad hoc mode. Mobility was provided by graduate students carrying the devices as they wandered a floor in an office building. We used two different techniques to implement a location service. The first was to simply have the user manually input their current location as they moved. The second was to perform rough localization using signal strength readings from 802.11 Access Points at known locations.

We tested our systems using two simple VN applications. The first maintained a counter in each region. Clients could increment and decrement the counter at will. We verified that even as clients failed and moved, the counters within continually-populated regions remained consistent over time. The second sample application is the traffic light described in Section 4. Again, we verified that the light performed as specified even as the set of client nodes in the region of interest changed.

3.3 Simulator Deployment

We examined more extensive deployments using a custom-built discrete-event, packet-level simulator that could run the deployment code with only minimal modifications.[2] The simulator uses a simple distance threshold-based criteria for determining who receives each message. It does not directly model packet collisions, but does allow the user to supply a probablistic message loss parameter. Mobility of the nodes in the simulation can be described by the traces produced by the popular `set-dest` tool included in the standard `ns-2` distribution. Using the simulator, we validated our VNLayer—running a simple geographic broadcast application—on deployments as large as 100 nodes moving over a geographic space divided into 25 regions.

3.4 Future Work and Code Availability

Our test deployments, as mentioned, are preliminary. We leave it as future work to perform more systematic experiments and produce detailed empirical descriptions of the system performance. To date, we have developed a set of analysis tools—including a visualizer, which can produce real-time animations from simulation traces, and a log parser, which can process deployment logs to produce statistics on the behavior of the VNs—to facilitate this evaluation. The code for the implementation and simulator are available at the Virtual Infrastructure Project Homepage [2].

On the theoretical front, it remains for us to complete a

[2]The only required change is to update the message multiplexer's socket code to connect to the the sockets exposed by the simulator as oppose to the sockets connected to the Network Interface Card.

rigorous formal analysis of our Reactive VN VNLayer emulator algorithm—confirming that it does, indeed, guarantee the properties described above. This analysis will follow the general format as those conducted for prior VNLayer implementations.

4. EXAMPLE: VIRTUAL TRAFFIC LIGHT

To demonstrate the utility of the VNLayer with reactive VNs described in the previous section, we present an example application: a virtual traffic light, designed to coordinate client vehicles entering and leaving a road intersection. Each client can communicate with other clients using a local broadcast service and has access to information about the time and its current location. The virtual traffic light allows clients approaching an intersection to determine whether it is safe to proceed without hitting another vehicle (e.g., by viewing the "virtual traffic light" displayed on their in-vehicle computer).

In our solution, a VN plays the role of a *Virtual Traffic Light*, informing the client running on each physical node of the color of the traffic light in its direction. The VN maintains the safety property that, at any moment, only clients in a single direction see a non-red light, and the liveness property that a waiting client eventually sees a green light (in normal cases, within a small time bound).

Using the VNLayer described in Section 3, our Virtual Traffic Light implementation was easy to write, requiring fewer than 300 lines of Python code. The main *msgReceived* function of the traffic VN is below:

```
def msgReceived(self, msg):
   replyArray = []
   if msg.msgtype(msg) == "UPDATE":
      self.UpdateVehInfo(msg)
      self.UpdateLightState()
      reply = "STATUS^" + self._statusSTR_()
      replyArray.append(reply)
   return replyArray
```

Recall that reactive VNs do not have a clock, whereas the clients in our application do. To help our VN make progress, clients timestamp their messages with their clock times, which are then adopted by the VN as its best estimate of the current time.

Clients periodically broadcast their current time, position, and heading in *UPDATE* messages to their local VN, the receipt of which prompts the VN to perform state updates and possibly change the color of some of the roads' lights, as described in the *msgReceived* function.

Specifically, the *UPDATE* messages allow the traffic VN to maintain information about the current system time and how many clients are incoming to the intersection in each direction. It then uses this state information to determine which directions get which color lights (the logic for these decisions is encapsulated in *UpdateLightState*). The VN broadcasts a *STATUS* message to local clients to describe the light color for each road direction.

In *UpdateLightState*, once the VN sees, based on its estimate of the local time, that a predetermined minimum "green time" has passed after the start of a green light for a direction (or it sees that no more vehicles are left in the direction with the green light), it changes the direction's light color to yellow. Similarly, if the VN sees that a predetermined minimum "yellow time" has passed after the start of a yellow light, it sets the light red and assigns a new green direction. In our implementation, *UpdateLightState* uses a fair algorithm, cycling through each populated direction in a round robin fashion, choosing them in order. If the light has just been initialized, the function returns the direction with the most vehicles, or a random direction if more than one direction has the maximum number of vehicles.

Whenever a client receives a *STATUS* message from the traffic VN, it parses the message to determine the color of the light in its direction, and then displays it.

We performed a preliminary evaluation of this application by deploying it on a small number of HP iPAQ hand-held computers. We used the Virtual Traffic Light application to control "traffic flow" in a narrow hallway connecting three different common areas on a floor of an office building. It performed as expected in this initial test, though we look forward to attempting larger scale deployments.

5. APPLICATIONS

Here we describe some applications faciliated by the VNLayer abstraction.

5.1 Basic communication and storage services

Geographical routing: A fundamental service for VNLayers, useful for many applications, is region-to-region routing (a form of *geographical routing*, also known as *geocast*). In [10], we present a self-stabilizing algorithm to implement geocast routing for timed stationary VNs using a persistent greedy depth-first search (DFS) routing algorithm that runs over the VNLayer. A similar algorithm could be used over reactive VNs.

Location management: A *location service* in an ad hoc network allows any client to discover the location of any other client using only its identifier. A location management scheme using timed stationary VNs is described in [10]. VNs serve as "home locations" [3, 11, 12] for clients. Each client's id hashes to the name of a VN, which serves as the client's home location, and is responsible for keeping track of the client's physical location. Clients keep their home locations informed of their whereabouts (using geocast), and other nodes can query those same home locations for the information. This, or even a low-stretch scheme, should be adaptable to reactive VNs.

End-to-end message routing: Another basic service in ad hoc networks is end-to-end IP-style message routing. Such a service should be easy to provide, given a geocast and a location service.

Data repositories: An auxiliary service that is useful for planning and control applications is a *data repository service*. Such a service can allow a VN to maintain a database of information about conditions in its local region and other regions. Resiliency can be built in by using techniques already designed for static but failure-prone networks, such as

automatically backing up data at neighboring VNs or sending data to a central, reliable location.

5.2 Robot motion coordination

In [13], we demonstrate a simple algorithm that uses timed stationary VNs to solve a fundamental motion coordination problem, namely, uniformly positioning mobile nodes (robots) on a known differentiable curve in the plane. The VNs act as controllers, managing movement of CNs in their own regions of the plane. The paper also demonstrates a simple "emulator-aware" approach to maintaining VNs: a VN attempts to keep itself alive by using local population density as one criterion for determining CN target destinations. The approach could be extended to take into account more client or network factors and even to provide active recruitment. Other coordination applications that could benefit from the use of a VNLayer include:

Intelligent-highways: We can extend the Virtual Traffic Light application into future intelligent-highway applications. Such applications will need to conduct a variety of activies, including collecting data (e.g., about traffic patterns), alerting cars about road hazards (e.g., accidents or arriving emergency vehicles), and providing advice and control. For example, a protocol may suggest less-congested alternative routes, or may emulate the functions of virtual traffic lights at intersections having no real traffic lights.

Air-traffic control (ATC): In current air-traffic control applications, ground-based human controllers coordinate the activities of aircraft. Future aircraft systems may allow aircraft greater autonomy in choosing flight plans dynamically [1], which will lead to a corresponding greater need for coordination protocols to resolve conflicting requirements for airspace. In areas where no human controllers are available, these protocols must be *distributed*, running on computers on board the aircraft, but must also emulate sensible, centralized air-traffic-control policies that are compatible with those enforced by ground controllers. In the scenario we envision a VN acting as an air-traffic controller, planning the trajectories of aircraft in its vicinity so as to avoid conflicts.

6. REFERENCES

[1] Blueprint for NAS. FAA, Office of System Architecture and Investment Analysis.

[2] Virtual infrastructure project homepage. http://theory.csail.mit.edu/tds/vi-project/index.html.

[3] I. Abraham, D. Dolev, and D. Malkhi. LLS: A locality aware location service for mobile ad hoc networks. In *Proceedings of the DIALM-POMC Joint Workshop on Foundations of Mobile Computing*, pages 75–84, Philadelphia, PA, October 2004.

[4] J. Beal. Persistent nodes for reliable memory in geographically local networks. Technical Memo AI Memo 2003-011, MIT AI Lab, Cambridge, MA, April 2003.

[5] G. Chockler, M. Demirbas, S. Gilbert, C. Newport, and T. Nolte. Consensus and collision detectors in wireless ad hoc networks. In *Proceedings of the Twenty-Fourth Annual Symposium on Principles of Distributed Computing (PODC 2005)*, pages 197–206, Las Vegas, Nevada, July 2005.

[6] S. Dolev, S. Gilbert, L. Lahiani, N. Lynch, and T. Nolte. Timed virtual stationary automata. In *9th International Conference on Principles of Distributed Systems (OPODIS 2005)*, December 2005. Also, Technical Report MIT-LCS-TR-979a, MIT CSAIL, Cambridge, MA 02139, August 2005.

[7] S. Dolev, S. Gilbert, L. Lahiani, N. Lynch, and T. Nolte. Timed virtual stationary automata for mobile networks. In *Allerton Conference 2005: 43rd Annual Allerton Conference on Communication, Control, and Computing*, page 323, Champaign-Urbana, IL, September 2005. Invited paper.

[8] S. Dolev, S. Gilbert, N. A. Lynch, E. Schiller, A. A. Shvartsman, and J. L. Welch. Virtual mobile nodes for mobile ad hoc networks. In R. Guerraoui, editor, *18th International Symposium on Distributed Computing (DISC 2004), Trippenhuis, Amsterdam, the Netherlands, October, 2004*, volume 3274 of *Lecture Notes in Computer Science*. Springer, December 2004. Also Technical Report MIT-LCS-TR-937, MIT CSAIL, Cambridge, MA 02139, 2004.

[9] S. Dolev, S. Gilbert, E. Schiller, A. A. Shvartsman, and J. Welch. Autonomous virtual mobile nodes. In *DIAL-M-POMC 2005: Third Annual ACM/SIGMOBILE International Workshop on Foundation of Mobile Computing*, pages 62–69, Cologne, Germany, September 2005. Also Technical Report MIT-LCS-TR-992, MIT CSAIL, Cambridge, MA, 2005.

[10] S. Dolev, L. Lahiani, N. Lynch, and T. Nolte. Self-stabilizing mobile node location management and message routing. In *7th International Symposium on Self Stabilizing Systems (SSS 2005)*, Barcelona, Spain, October 2005. Also, Technical Report MIT-LCS-TR-999, MIT CSAIL, Cambridge, MA, August 2005.

[11] Z. Haas and B. Liang. Ad hoc mobility management with uniform quorum systems. *IEEE/ACM Trans. on Networking*, 7(2):228–240, April 1999.

[12] J. Li, J. Jannotti, D. S. J. DeCouto, D. R. Karger, and R. Morris. A scalable location service for geographic ad hoc routing. In *Mobicom 2000: The Sixth International Conference on Mobile Computing and Networking*, pages 120–130, Boston, Massachusetts, August 2000.

[13] N. Lynch, S. Mitra, and T. Nolte. Motion coordination using virtual nodes. In *44th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC 2005)*, Seville, Spain, December 2005. Also Technical Report MIT-LCS-TR-986, MIT CSAIL, Cambridge, MA 02139, April 2005.

[14] M. Steenstrup. *Ad Hoc Networking*, pages 75–138. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[15] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.