Virtual Infrastructure for Collision-Prone Wireless Networks

(Extended Abstract)

Gregory Chockler
IBM Research
chockler@il.ibm.com

Seth Gilbert
EPFL IC
seth.gilbert@epfl.ch

Nancy Lynch
MIT CSAIL
lynch@mit.edu

ABSTRACT

Wireless ad hoc networks pose several significant challenges: devices are unreliable; deployments are unpredictable; and communication is erratic. One proposed solution is *Virtual Infrastructure*, an abstraction in which unpredictable and unreliable devices are used to emulate reliable and predictable infrastructure. In this paper, we present a new protocol for emulating virtual infrastructure in collision-prone wireless networks. At the heart of our emulation is a *convergent history agreement* protocol that tolerates lost messages and crash failures. It is designed specifically for ad hoc deployments, for example, the set of participants is *a priori* unknown. The convergent history agreement protocol is quite efficient, as each agreement instance completes in a constant number of communication rounds, and the size of the messages is constant, independent of the length of the execution. Building on the convergent history agreement protocol, our virtual infrastructure emulation introduces only constant overhead per virtual round emulated. We believe that the techniques developed in this paper help to bring virtual infrastructure one step closer to a reality.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*wireless communication*; C.4 [**Performance of Systems**]: Fault Tolerance; D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*

General Terms

Algorithms, Reliability

Keywords

Fault Tolerance, Mobile Ad Hoc Networks

PODC'08, August 18–21, 2008, Toronto, Ontario, Canada.

1. INTRODUCTION

There are several significant challenges associated with developing algorithms for wireless ad hoc networks: wireless devices are often unreliable and crash-prone; wireless devices are often mobile and move in an unpredictable manner; and wireless communication is unreliable due to channel contention, collisions, and other forms of interference. All of these challenges can be lessened by the deployment of a reliable, fixed infrastructure. For example, on most university campuses, wireless 802.11 base-stations are deployed to allow for reliable connectivity among oft-mobile devices. Unfortunately, it is often impractical due to logistical or cost-related concerns to deploy such a fixed infrastructure.

Virtual infrastructure appears to be a promising alternative (see, e.g., [11–15, 35]). The idea underlying a *virtual* infrastructure is to emulate a fixed infrastructure in an ad hoc environment in which real hardware-based infrastructure is unavailable. Implemented as middleware, for example, the virtual infrastructure abstraction provides the programmer with the illusion of a reliable, static infrastructure consisting of *virtual nodes* deployed at regular locations throughout the world. Virtual nodes are *reliable*, in that they remain active as long as any mobile device is nearby, and they are *predictable*, in that they are immobile.

As has been demonstrated elsewhere, virtual infrastructure is instrumental in implementing a wide variety of applications, such as reconfigurable atomic memory [13], routing [12,16], location / tracking services [11,16,34,36], mobile robot coordination [4, 27], and air-traffic control [3]. Ongoing projects include IP address allocation in ad hoc networks [47], overlay-based routing [40], IPv6 routing [17], and highway traffic optimization [4]. In general, virtual infrastructure appears quite promising as a technique for simplifying the development of algorithms for ad hoc networks.

The promise of virtual infrastructure is clear; in this paper, we address the question of *how to emulate virtual infrastructure with low overhead in unreliable, collision-prone wireless networks.* At the heart of our proposed virtual infrastructure emulation is a new building block which we call *convergent history agreement* (CHA), an iterated agreement protocol tailored to the complexities of wireless networks. We present an efficient protocol for solving CHA in collision-

*This work was supported in part by the following: MICS, AFOSR A9550-04-1-0121 and A9550-08-1-0159, NSF CCF-0726514 and CNS-0715397. The full version of this paper [19] is available at:
`http://groups.csail.mit.edu/tds/vi-project/biblio.html`

prone wireless networks, and discuss how to transform our CHA protocol into a virtual infrastructure emulation.

1.1 Wireless Communication

In this section, we highlight some important aspects of the communication model considered in this paper; a complete description is provided in Section 2.

We assume that communication is synchronous (i.e., "slotted"), and that in each round of communication, a node may choose to broadcast or listen on the channel. Eventually, during intervals in which the network is well-behaved, nearby nodes can communicate, as long as there is no contention on the channel. During intervals when the network is badly behaved, however, all messages may be lost.

Collision Detectors.

We assume that nodes have a limited collision detection capability (without which consensus is impossible, see e.g., [7, 8]). The justification for collision detection comes from recent work [10,37,46] which argues that collisions can often be detected with reasonable precision by nodes listening on the channel using carrier sensing and other techniques. We capture collision detection guarantees using the approach introduced in [8]. We assume that collision detectors are:

- *complete* (Property 1): There are no false negatives; when a collision occurs, the detector reports a collision.
- *eventually accurate* (Property 2): There are eventually no false positives; eventually, the collision detector reports a collision only when a message has been lost.

Contention Management.

In a wireless network, one cause of message loss is high channel contention. Typically, wireless ad hoc networks make use of protocols such as exponential backoff to reduce contention. Often "contention management" is integrated into the underlying protocol, resulting in a complicated analysis and an intermingling of safety and liveness concerns.

In this paper, we separate the issue of contention management from the problem of emulating virtual infrastructure. The emulation algorithm described in this paper is *compatible* with a wide class of contention managers. The contention manager designates nodes as *active* (i.e., enabled to broadcast) or *passive*; it guarantees that eventually there is only one active node (Property 3).

In practice, contention managers are typically implemented using randomized back-off protocols. The problem of designing efficient back-off protocols has been well studied (e.g., [18, 31,32,45]), and is not the focus of this paper; we believe even a simple exponential back-off scheme to be sufficient.

1.2 Virtual Infrastructure

A *virtual infrastructure* consists of a set of deterministic *virtual nodes* distributed throughout the network, each of which resides at a fixed location. The virtual nodes interact with *clients*[1], and behave, from the clients' perspective, just like any other (real) wireless device.

Clients and virtual nodes communicate using a virtual broadcast service. As in the underlying system, virtual communication is "wireless," in that it remains collision prone, and proceeds in rounds. Both the clients and the virtual nodes are equipped with eventually accurate, complete collision detectors and a contention manager.

We emphasize again that a system containing virtual nodes appears, from a client's perspective, equivalent to a system in which each virtual node is replaced with a reliable, immobile real device.

1.3 Key Ideas

The basic idea underlying our implementation of virtual infrastructure is that unreliable mobile devices cooperate to emulate reliable virtual nodes. Whenever a mobile device enters a virtual node's region, it joins the emulation of the virtual node and becomes a *replica*. When a mobile device leaves a virtual node's region, it ceases to participate in the emulation. The key challenge, then, is maintaining the consistency of the replicas as mobile nodes join and leave, despite the possibility of unreliable communication.

Convergent History Agreement.

In order to maintain consistency, the replicas execute a sequence of agreement instances, one per virtual round. We refer to this problem of repeated agreement, which is closely related to the replicated-state-machine paradigm, as *convergent history agreement*.

Each instance of the agreement protocol is associated with the emulation of one virtual round. The replicas try to agree on a history of the virtual node up to (and including) the virtual round associated with that instance. When an agreement instance completes successfully, all the replicas agree upon the same history, allowing for a consistent emulation.

When an agreement instance does *not* complete successfully, however, some disagreement may arise. In this sense, convergent history agreement is subtly different from classical consensus in that it allows for a *limited* amount of disagreement. Every (correct) node produces an output for every instance; however, for some instances, some subset of the nodes may output $\perp$, indicating that no decision was reached for that instance. Thus, after any given virtual round, different nodes may disagree on which instances have produced a decision and which instances have produced $\perp$. Any time that a node *does* successfully produce an output, it must resolve all the prior undecided agreement instances in a consistent fashion[2]. This limited disagreement is of key importance for the efficiency of the emulation.

Indeed, one of the main technical challenges in this paper is balancing the trade-off between the consistency of the replicas and the efficiency of the emulation. In terms of efficiency, we are particularly interested in the size of the messages and the number of communication rounds required by the emulation; together, these define the overhead introduced by the virtual infrastructure emulation.

In Section 3, we formally define the problem of *convergent history agreement* (CHA), specifying the precise consistency

[1]In order to distinguish between the real mobile nodes, on which the emulator is executed, and the "abstract" mobile nodes, which interact with virtual nodes and execute the user's program, we refer to the latter as "clients."

[2]To illustrate, consider the following scenario: There are two nodes p_i and p_j that are unable to communicate due to interference. Node p_i outputs a decision and fails. In this case, p_j is required to behave in a manner consistent with this unknown decision! The natural solution would be for p_i to output a decision only after receiving an acknowledgment from p_j. However, p_i cannot exchange messages with every other node, since only one message can be sent on the channel at a time, and hence such communication would take $\Theta(n)$ time, which would be inefficient.

requirements. We then present a CHA protocol that uses only a constant number of communication rounds for each instance, and sends only constant-sized messages, independent of the number of nodes or the length of the execution.

Emulation protocol.

While the problem of convergent history agreement is at the heart of implementing a single virtual node, many further issues arise in the emulation of a complete virtual infrastructure. There are two main problems that arise. First, the protocol messages sent by emulators for nearby virtual nodes may interfere with each other, causing unexpected collisions. Second, the emulation protocol must support communication between neighboring virtual nodes. This latter scenario is difficult in that it involves carefully synchronizing the agreement protocols to maintain the consistency of each virtual node. In order to cope with these challenges, each virtual round is simulated using two separate instances of the CHA protocol, one for virtual nodes that are broadcasting, and another for virtual nodes that are listening. The virtual node emulations are carefully scheduled to ensure that interference between neighboring instances does not overly disrupt the emulation. In Section 4, we lay out the entire emulation protocol and outline some of the issues that arise.

1.4 Summary of Contributions

1. Convergent History Agreement: We define the problem of convergent history agreement, a variant of the *replicated-state-machine approach*, which provides a sufficient level of consistency for implementing virtual infrastructure, while at the same time yielding an efficient implementation. We present a new algorithm for solving convergent history agreement, the first replicated-state-machine implementation that is suited for a collision-prone wireless network.

It uses a novel strategy, inspired by three-phase commit [41, 42], to ensure consistent outputs despite collisions, lost messages, and crash failures. And it uses only a constant number of rounds per instance of agreement, despite the need to coordinate over a contention-prone channel with up to n nodes. Every protocol message is constant sized, independent of n and the length of the execution.

2. Virtual Infrastructure Emulation: We describe an algorithm for emulating virtual infrastructure, based on the convergent history agreement protocol presented in Section 3. In addition to the issues associated with a single virtual node, it also addresses the problem of coordinating nearby virtual nodes in such a way as to reduce the interference and enable communication between virtual nodes.

The emulation algorithm ensures consistency among the replicas, tolerating collisions, lost messages, and crash failures. The resulting protocol copes with ad hoc deployments in that it can tolerate a changing set of participants, and does not require any knowledge of the number of participants; nor does it require that the participants have unique identifiers. In addition, the contention management is decoupled from the main algorithm, allowing for a separation of liveness and safety concerns.

Finally, the emulation protocol is efficient in the sense that it requires only a fixed number of rounds to implement a single virtual round of the virtual infrastructure, independent of the number of replicas and the length of the execution. (It depends only on the density of the virtual node deployment.) Moreover, the protocol messages are all of constant size when the system is stable.

The full, detailed presentation of our model, algorithms, and proofs can be found in [19]; due to space limitations, we provide in this paper only an extended abstract.

1.5 Other Related Work

There is a close connection between the problem of *convergent history agreement* and the replicated-state-machine paradigm (e.g., [24, 25, 39]). Replicated-state machines have been implemented under a variety of failure models and environments (e.g., [6, 23, 26]). These implementations do not address the unique challenges posed by the wireless ad hoc networks such as channel contention, unpredictable message loss, and unknown participants. For example, most such protocols require at least a majority of the nodes to send messages; in a wireless network this creates unacceptable channel contention and long delays.

Convergent history agreement is also similar to another interesting line of research on *continuous consensus*, in which each node maintains an up-to-date *core* of information about the past in such away that all the cores are identical. See, for example, [28–30]. As in the case of replicated-state-machine protocols, the continuous consensus protocols tend to involve significant amounts of all-to-all communication, which is infeasible in the context of wireless networks.

There is also a connection between our new CHA protocol and classic *three-phase commit* (3PC) protocols [41, 42]. The 3PC protocols, however, take a somewhat different approach to recovering from network misbehavior. There are also some similarities to the "Enhanced Three Phase Commit" protocol of [22]. Another interesting connection is to query-abortable objects [1] which also capture the notion that some agreement instances may fail.

Of late, there has been much work on high-level programming languages and region-based abstractions for ad hoc networks, particularly sensor networks, e.g., [33, 43, 44]. Much of this work is complementary to the work on virtual infrastructure: better programming languages are essential to simplifying the task of developing software for ad hoc networks; providing reliable virtual infrastructure with strong consistency guarantees can simplify the programming paradigm, regardless of the language employed.

2. THE SYSTEM MODEL

In this section, we present the underlying system model. The model in this paper is derived from [8, 9].

We consider in this paper a wireless network that consists of a fixed, but *a priori* unknown, collection of mobile nodes $P = \{p_1, p_2, \ldots, p_n\}$. At any given time, a node resides at a location in the plane, and its velocity is bounded by v_{max}. Each node receives periodic updates as to its location from a GPS, or some other variety of location service (e.g., [38]). The number of nodes is unknown, and nodes do not have unique identifiers. Nodes can fail by crashing at any point during the execution of the algorithm.

Communication is synchronous ("slotted"), based on a variant of the quasi-unit-disk model of communication: two nodes within broadcast radius R_1 of each other are able to communicate; two nodes within interference radius R_2 interfere with each other. (The virtual broadcast service that operates in the virtual infrastructure abstraction has its own virtual broadcast radius and virtual interference radius.)

Communication is prone to *collisions*, which can occur for arbitrary and unpredictable reasons. As a result of a collision, each node can fail to received an arbitrary subset of messages that were broadcast in a round. Moreover, collisions may affect nodes in a non-uniform way: a message may be received by some nodes, but not others.

We assume that, eventually, arbitrary collisions cease and all collisions are caused by contention on the channel. That is, there exists a round r_{cf} such that in every round $r \geq r_{\mathsf{cf}}$: if some source p_i broadcasts a message m in round r, and (i) some non-failed receiver p_j is within distance R_1 of p_i, and (ii) no node within distance R_2 of p_j broadcasts in round r, then p_j receives the message m.

We assume that every node $p_i \in P$ is augmented with a *collision detector* that delivers a collision notification $\pm$ to p_i when it fails to receive a message. We assume that the collision detectors are in the class $\diamond\mathcal{A}$-$\mathcal{C}$, as defined in [8], meaning they satisfy the following properties:

PROPERTY 1 (COMPLETENESS). *For every round r, if p_i does not receive some message that was broadcast in round r by a node within distance R_1 of p_i, then p_i detects a collision in round r.*

PROPERTY 2 (EVENTUAL ACCURACY). *For every execution, there exists a round r_{acc} such that for every round $r \geq r_{acc}$, and every node $p_i \in P$: if p_i detects a collision in round r, then at least one message that is not received by p_i was broadcast by a node within distance R_2 of p_i.*

In order to model random backoff protocols, we assume the existence of a set of *contention managers*, one for each virtual node being implemented. In each round, each mobile node can *contend* for one of the contention managers, requesting access to the broadcast channel in that region. The contention manager then provides advice to each node as to whether it should be active—that is, broadcast—in a round. The goal is to capture the guarantees provided by a simple back-off protocol, such as exponential back-off. Thus, a contention manager can be implemented using standard techniques (see, e.g., [18, 31, 32, 45]).

Each contention manager is associated with a specific location ℓ, the location of some virtual node. It reduces contention among contending nodes that are close to ℓ, while ensuring that at least one contending node is allowed to broadcast. For the purpose of Section 3, when we refer to nodes that are "close to ℓ," we mean within distance $R_1/2$, as this ensures that all contending nodes can communicate with each other. For the purpose of Section 4, by contrast, we mean within distance $R_1/4$ of ℓ; this turns out to be the size of the region in which we implement virtual nodes. In both cases, the fact that the contention-management region is smaller than the broadcast radius should simplify the contention manager's implementation.

For the problem of "convergent history agreement," discussed in Section 3, a relatively simple *leader election* contention manager is sufficient:

PROPERTY 3 (LEADER ELECTION). *If only nodes within distance $R_1/2$ of ℓ contend for contention manager C_ℓ, then:*

1. *Eventually, at most one node is advised by C_ℓ to be active in every round.*
2. *If any correct node contends in every round, then eventually, some correct node $p_j \in P$ is advised to be active in every round.*
3. *C_ℓ advises a node p_i to be active in round r only if p_i contends for round r.*

For emulating virtual infrastructure, and in general, for tolerating a more dynamic environment, a somewhat more involved contention manager is needed, since there may be no correct node that remains in the region forever. We briefly discuss the properties of this contention manager in Section 4.2, where we describe the conditions under which a virtual node makes progress.

A note on eventual properties.

Both Properties 2 and 3, include properties that hold from some point onwards. This is simply a formal convention meaning that the property holds for *long enough*. In reality, there may be alternating periods of stability and instability. For the purpose of this paper, these properties need only hold for a fixed small number of rounds; the progress guarantees hold during these good intervals. (Safety guarantees hold regardless, throughout the execution.) While it is unrealistic to assume stability forever, it is quite realistic to assume that there are short stable intervals.

3. CONVERGENT HISTORY AGREEMENT

At the heart of the virtual infrastructure emulation is a new agreement protocol for wireless networks that are prone to collisions and lost messages. This protocol is, to the best of our knowledge, the first instantiation of the replicated-state-machine paradigm for collision-prone wireless networks.

3.1 Overview

We begin with some motivation for the problem of convergent history agreement. In the context of implementing virtual infrastructure, we replicate each virtual node at a set of nearby mobile nodes, thus ensuring fault tolerance. Typically, when replicating a service, the participating replicas execute a consensus protocol each time the state of the service is modified, agreeing on the new state of the service. In this case, the participating replicas need to agree on the result of each virtual round, thus determining a consistent behavior for the virtual node in that round. (In particular, the replicas need to agree on the set of messages that the virtual node receives in each round.) This general strategy describes the replicated-state-machine approach for implementing a virtual node, and is the basis for prior virtual infrastructure constructions (e.g., [11–14]).

Unfortunately, as was shown in [8], it is impossible in the presence of ongoing collisions to solve consensus efficiently, i.e., in a constant number of communication rounds. It is only after the system stabilizes that efficient consensus is feasible. To overcome this fact, we allow some disagreement among the replicas, particularly during periods when the network is unstable. A key observation is as follows: unlike in the typical replicated-state-machine approach, the replicas do not need to agree in each round, as the internal state is not externally visible; it is only necessary that the replicas agree during a virtual round in which messages sent by the virtual node are delivered to another node. We capture this slightly weaker notion of agreement in the problem of *convergent history agreement*.

A key feature of convergent history agreement is that, instead of outputting a value for each agreement instance,

each node outputs a *history*. In this way, we can capture the notion that:

1. In any given round, there may be some (limited) disagreement. That is some nodes may output a history, while others may output $\perp$.

2. But, the outputs must converge to a single history that is consistent with all prior outputs.

3.2 Problem Definition

In this section, we define the problem of *convergent history agreement* (CHA). In order to focus on the agreement problem at the heart of implementing virtual infrastructure, we restrict our attention to a simpler, more static environment. For the remainder of Section 3, we assume that all n nodes remain always within distance $R_1/2$ of some fixed location ℓ, and that at least one is correct. We assume that there is a leader-election contention manager C_ℓ located at location ℓ. We postpone discussing the issues that arise in a more dynamic setting to Section 4.

Let V be a totally-ordered domain of values (exclusive of the special symbol $\perp$) that serve as inputs to the CHA protocol. A *history* $h : \mathbb{N} \to V \cup \{\perp\}$ is a function mapping each non-negative integer to either a value, or the special symbol $\perp$. Denote by $H(V)$ the set of histories over V.

An execution of CHA consists of a (possibly infinite) sequence of instances $\langle k_1, k_2, \ldots\rangle$. For each instance k, each non-failed node $p_j \in P$ proposes an input value $v_{j,k} \in V$. After each non-failed node has proposed an input for instance k, then each $p_j \in P$ produces an output $h_{j,k} \in H(V) \cup \{\perp\}$; that is, instance k outputs either a history, or the special symbol $\perp$ (indicating that no history is available). Notice that for a given instance, some nodes may output a history, while others output $\perp$. For a given set of instances $\langle k_1, k_2, \ldots\rangle$, outputs $h_{*,*}$ must satisfy the following requirements:

1. **Validity:** For every instance k, for every output $h_{*,k} \neq \perp$, for every $k' \leq k$: if $h(k')_{*,k} \neq \perp$, then for some node $p_j \in P$, the proposal $v_{j,k'} = h(k')_{*,k}$. That is, for every history output, every value included in the history was proposed for the corresponding instance.

2. **Agreement:** For every pair of instances $k_1 \leq k_2$, for every pair of nodes $p_i, p_j \in P$: if output $h_{i,k_1} \neq \perp$ and if output $h_{j,k_2} \neq \perp$, then for every $k \leq k_1$, $h(k)_{i,k_1} = h(k)_{j,k_2}$. That is, every pair of histories agree on a common prefix of the history.

3. **Liveness:** There exists an instance k_{st} such that for every instance $k \geq k_{st}$, for every non-failed node $p_j \in P$: (1) $h_{j,k} \neq \perp$, and (2) for every $k' \in [k_{st}, k]$, output $h(k')_{j,k} \neq \perp$.

We say that history h includes instance k if $h(k) \neq \perp$. Notice that *agreement* does not require that every two nodes produce the same output in every instance; it requires only that *if* a node outputs a history, instead of $\perp$, then it agrees with all other histories.

3.3 CHA and Virtual Nodes

Notice that CHA captures the problem of agreement that lies at the heart of implementing a virtual node. Aside from the issues associated with joining and leaving, an algorithm for CHA can be used to instantiate a virtual node. Each mobile node executes two components: the program of the *client* that wants to interact with the virtual node, and the CHA algorithm that emulates the virtual node in a replicated manner.

Each instance k of CHA is associated with virtual round k. A virtual round begins when the clients broadcast their messages for that virtual round. The associated CHA instance then attempts to agree on which messages the virtual node should receive. That is, for virtual round k, a node p_j includes in its proposal the set of messages that p_j believes should be delivered to the virtual node in that virtual round.

When a CHA instance outputs a history at node p_j, then p_j can calculate the state of the virtual node. For an instance k' that is included in the history, p_j simulates the virtual node receiving the messages included in the proposal; for an instance k' that is not included in the history (i.e., for which the history includes a $\perp$), replica p_j simulates the virtual node detecting a collision. By the Agreement Property of CHA, we can conclude that whenever a CHA instance outputs a history, it calculates a state consistent with every other replica. The Validity Property of CHA ensures that the virtual node history is consistent with the proposals, i.e., the real messages that the clients sent.

When a CHA instance does not output a history, i.e., when it outputs $\perp$, the replica instructs its co-located client to simulate detecting a collision. Thus, in this case, the client cannot determine whether or not the virtual node sent a message, and hence the virtual node performs no externally visible action.

Eventually, by the Liveness Property, every instance outputs a history, and every instance is included in every history. This ensures both that eventually the virtual node makes progress, and also that the collision detectors are both complete and eventually accurate.

In Section 4 we discuss further the issues that arise when using CHA to implement a full virtual infrastructure.

3.4 The CHA Protocol

We now describe in detail the CHAP protocol in Figure 1 that solves CHA. It executes each agreement instance using only a constant number of rounds, and uses only constant-sized messages, independent of the length of the execution. (By contrast, a naïve solution might include the entire history in every message.) Note also that the algorithm tolerates crash failures. In Section 3.5, we discuss the issue of reducing local space usage via garbage collection.

Colors.

For each node p_j, the $status_j$ array stores the color that p_j assigns to each instance. There are four possible colors: *red* $<$ *orange* $<$ *yellow* $<$ *green*. The color reflects each node's local knowledge about *the other nodes'* knowledge regarding the status of the instance. The key property guaranteed by the algorithm is as follows:

PROPERTY 4. *For every instance k, no two nodes choose colors for instance k that differ by more than one shade (see Lemma 5).*

Thus, if some node designates an instance as green, then every node designates it as either green or yellow; if some node designates an instance as red, then every node designates it as either red or orange.

Figure 1: Convergent History Agreement Protocol (CHAP)

```
1  Interface:
2     Input propose(k)_j, returns proposal v ∈ V for instance k from p_j
3     Input cm-wakeup()_j, returns contention manager advice for p_j
4     Output output(h)_j, returns output h ∈ H(V) ∪ {⊥}

5  Data Structures:
6     k, prev-instance ← 0
7     status[] ← ⟨green, green, ..., green⟩
8     ballot[], an array of:
9        prev-instance, initially 0
10       v ∈ V ∪ {⊥}, initially ⊥
11    phase ∈ ⟨ballot, veto-1, veto-2⟩, initially ballot

12 bcast()_i
13    case phase = ballot:
14       k ← k + 1
15       v ← propose(k)_j
16       b ← ⟨v, prev-instance⟩
17       if cm-wakeup()_j then
18          return b
19       else return ⊥
20    case phase = veto-1:
21       if status[k] = red then
22          return ⟨veto⟩
23       else return ⊥
24    case phase = veto-2:
25       if status[k] = red or orange then
26          return ⟨veto⟩
27       else return ⊥

28 recv(M)_i
29    case phase = ballot:
30       if (M = ∅) or (± ∈ M) then
31          status[i] ← red
32       else ballot[i] ← min(M)
33    case phase = veto-1:
34       if (veto ∈ M) or (± ∈ M) then
35          status[i] ← min(orange, status[i])
36    case phase = veto-2:
37       if (veto ∈ M) or (± ∈ M) then
38          status[i] ← min(yellow, status[i])
39       if (status[i] ∉ {red,orange} then
40          prev-instance ← k
41       history ← calculate-history(k, prev-instance, ballot)
42       if (status[k] = green) then
43          output(history[k])_j
44       else
45          output(⊥)

46 function calculate-history(instance, prev, ballot)
47    ∀ k ≥ 1 do: history[k] ← ⊥
48    for k = instance downto 1 do
49       if (k = prev) then
50          history[k] ← ballot-array[k].v
51          prev ← ballot[k].prev-instance
52       else
53          history[k] ← ⊥
54    return h
```

If a node designates an instance as either green or yellow, we say that it considers the instance to be *good*. Throughout, the $prev\text{-}instance_j$ variable maintains the most recent good instance, i.e., the most recent instance that p_j has designated as either yellow or green.

Ballots.

For each node $p_j \in P$, the $ballot_j$ array stores the ballot selected by p_j for each instance k. Node p_j can use the stored ballots to calculate possible histories; at any given time, there may be multiple possible histories that can be legally computed by p_j.

Each entry of the ballot array contains two fields: a value $v \in V \cup \{\perp\}$, and a pointer *prev-instance* that identifies an earlier instance. A particular history can be computed by following the *prev-instance* pointers backward through the ballot array, adopting the value in each case and then assigning all other instances to $\perp$. For example, consider beginning at some instance k_9. We can then calculate:

$$\begin{aligned} k_8 &= ballot[k_9].prev\text{-}instance, \\ k_7 &= ballot[k_8].prev\text{-}instance, \end{aligned}$$

and so on, until a sequence of instances $\langle k_1, k_2, \ldots, k_9 \rangle$ has been determined. At this point, a history h can be defined as follows:

$$h(k) \leftarrow \begin{cases} ballot[k].v & \text{if} \quad k \in \{k_1, \ldots, k_9\} \\ \perp & \text{if} \quad k \notin \{k_1, \ldots, k_9\} \end{cases}$$

This is, in fact, precisely the calculation being performed by the calculate-history function (lines 46–54). It takes three inputs:

- *instance*, which identifies the "current" instance. In our example above, this is some ballot $k_{10} > k_9$. This instance may or may not be good.
- *prev*, which identifies the largest known good instance. In our example above, this is k_9. Notice that the *prev* pointer cannot be calculated using the *instance* input, as there is no guarantee that *instance* is a good instance.
- *ballot*, which contains a copy of the ballot array.

The calculation proceeds to decrement k from *instance* down to 1 (lines 48–53). Whenever $k = prev$, we reset *prev* to $ballot[k].prev\text{-}instance$ (line 51). Notice that in the process, we are computing exactly the set $\{k_1, k_2, \ldots, k_9\}$ of instances found in the example above by following the pointers backwards. Whenever we find a $k \in \{k_1, \ldots, k_9\}$, we set the $history[k] = ballot[k]$ (line 50). Otherwise, we set the $history[k] = \perp$ (line 53).

Intuitively, any history that is produced by beginning at a good instance and following the *prev-instance* pointers backward forms a possible history. Moreover, if two nodes begin calculating their history in the *same* good instance, then we can show that both nodes calculate the same history. By Property 4, we know that if some instance k is green, then every node designates instance k as good, and hence updates their *prev-instance* pointer; from this we can conclude that, in this case, they all calculate the same history.

Phases.

For each instance, the main algorithm consists of three rounds of communication, known as phases. During these three phases, each node makes two important decisions: it selects a ballot and it selects a color. Moreover, it must ensure that the choice of colors satisfies Property 4. Each instance begins with a ballot phase, and then continues with two veto phases. See Figure 2 for a summary of how colors are assigned. Throughout, every (correct) node contends for the contention manager C_ℓ located at location ℓ.

ballot	veto-1	veto-2	**Replica Color**	**Output?**
√	√	√	Green	History
√	√	X	Yellow	⊥
√	X	X	Orange	⊥
X	X	X	Red	⊥

Figure 2: Table indicating how a node responds to collisions in the different phases of the algorithm: ballot/veto-1/veto-2. A check (√) indicates that the node receives a message in that round, and no collisions or veto indications are received. An X indicates that the node does not correctly receive the message in that round.

The nodes begin with the ballot phase, in which each node first updates its instance counter k (line 14), and retrieves the proposal (line 15). It then assembles a ballot b, which consists of the proposal for instance k, along with the *prev-instance* pointer (line 16). The node then checks with the contention manager (line 17) and returns the ballot or $\perp$ based on its advice.

Each node p_j then receives its messages M for the ballot phase (lines 29–32). If p_j does not receive a ballot, or if a collision is detected, then the instance is designated as red. Otherwise, the node selects a ballot from the set M deterministically (say, ordering the ballots lexicographically) (line 32).

The nodes then proceed to the veto phases. In each veto phase, a node broadcasts a veto if the instance is no longer green (line 21, line 25). If a collision is detected: (a) in the veto-1 phase, the status is downgraded to orange (line 35); (b) in the veto-2 phase, the status is downgraded to yellow (line 38).

In any instance that is designated as green or yellow, i.e., if the instance is good, then the *prev-instance* pointer is updated to k, the current instance (line 40).

The history is calculated via the calculate-history function (line 41) and an output is produced: if the instance is green, the output is a history; otherwise, the output is $\perp$ (lines 43–45). We will argue that if the instance is designated green, then we can be sure that every node calculates the same history, and thus these outputs satisfy agreement. We argue in Section 3.6 that the outputs satisfy the CHA requirements.

3.5 Space usage

As currently specified, the local storage (and hence the local computation) is unbounded. While this does not affect the message size, it may still be undesirable. It is possible to modify the algorithm so as to garbage collect portions of the data structure that are no longer needed.

In order to achieve this, we could consider an alternative version of CHA agreement known as "checkpoint-CHA" in which each node outputs a checkpoint, along with the suffix of the history including every instance after the checkpoint (instead of producing the entire history as an output).

In this case, a node p_j can "garbage-collect" whenever a round is designated as green, keeping only (1) a pointer to the most recent green round, (2) the checkpoint up to and including that round, and (3) ballot/status entries that have occurred since that green round. Notice that in rounds that are not designated as green, the node *cannot* perform such garbage-collection, as there are multiple possible executions.

3.6 Proof Sketch

In this section, we argue that CHAP is correct. A key invariant is that no two nodes differ in their color designation by more than one shade. Stated in a slightly different way:

LEMMA 5. *Let p_j, p_r be two nodes, and k an instance. If p_j designates k as green, then p_r designates k as green or yellow. If p_j designates k as red, then p_r designates k as red or orange.*

PROOF (SKETCH):. Assume p_j designates k as green and p_r designates k as orange or red; then p_r broadcasts in the veto-2 phase, ensuring that p_j does not designate k as green, which is a contradiction. On the other hand, if p_j designates k as red, then it broadcasts a veto in the veto-1 phase, ensuring that every node designates it as orange or red. □

Next, we conclude from Lemma 5 and calculate-history that if some history includes instance k, then no node p_j designates instance k as red. This is important, as such a node p_j might not be able to reconstruct the history including instance k.

LEMMA 6. *Let h be a history output by p_r that includes instance k. Then no node designates k as red.*

From this we observe that if two histories include some instance k, then neither designates it as red, and thus both have identical values for ballot k.

LEMMA 7. *Let h, h' be two histories output by p_j and p_r (respectively) that both include instance k. Then $ballot[k]_j = ballot[k]_r$ and $h(k) = h'(k)$.*

Thus, we approach convergence: if two histories include instance k, then they agree on all prior instances.

LEMMA 8. *Let h, h' be two histories output by p_j and p_r (respectively) that both include instance k. Then for every $k' \leq k$, $h(k') = h'(k')$.*

PROOF (SKETCH):. Notice that if instance k_1 is included in h and h', and $k_2 = ballot[k_1].prev\text{-}instance_j$, then for every $k' : k_2 \leq k' \leq k_1$, $h(k') = h'(k')$ by the operation of calculate-history. The lemma then follows by backward induction. □

In order to show agreement, it remains only to show that histories will eventually include the same instance. Specifically, we show that if an instance is designated as green, then every history includes that instance.

LEMMA 9. *Assume that p_j designates instance k as green. Then $\forall k' \geq k$, history $h_{*,k'}$ includes instance k.*

Finally, combining Lemma 8 and Lemma 9, we conclude that the algorithm satisfies agreement:

THEOREM 10 (AGREEMENT). *Given histories $h_{*,k_1} \neq \perp$, $h_{*,k_2} \neq \perp$, $k_1 \leq k_2$: for all $k \leq k_1$, $h(k)_{*,k_1} = h(k)_{*,k_2}$.*

PROOF (SKETCH):. Since history h is output for instance k_1, we can conclude that instance k is designated as green by some node. Thus we conclude by Lemma 9 that both history h_{*,k_1} and h_{*,k_2} include instance k_1, and thus the result follows by Lemma 8. □

We now proceed to show that the protocol satisfies the liveness property.

LEMMA 11. *Eventually every instance is designated green by every node.*

From this it follows immediately that the liveness condition is met:

THEOREM 12 (LIVENESS). *There exists $k_{st} : \forall k \geq k_{st}$, $\forall p_j$: (1) $h_{j,k} \neq \pm$, (2) $\forall k' \in [k_{st}, k], h(k')_{j,k} \neq \perp$.*

PROOF (SKETCH):. By Lemma 11, every node eventually designates every instance as green, and thus for some k_{st} outputs a history for every instance $> k$. Fix some $k' > k_{st}$, and assume history h is output for instance k'. Fix some $k'' : k' \geq k'' > k$. It is easy to see that for every history h' output at the end of k'' includes instance k'', since instance k'' is designated as green. By Theorem 10, then, we conclude that history h includes instance k''. □

The validity condition is trivially seen to be true as all values in ballots originate as proposals:

THEOREM 13 (VALIDITY). *For every $h_{*,k}$, $\forall k' \leq k$: if $h(k')_{*,k} \neq \pm$, then $v_{j,k} = h(k')_{*,k}$ for some node p_j.*

Finally, notice that the CHA protocol introduces only constant overhead[3]:

THEOREM 14 (OVERHEAD). *Every instance of CHA is instantiated using a constant number of rounds, and every message is constant size.*

4. VIRTUAL INFRASTRUCTURE EMULATION

In this section we present an overview of the complete algorithm for emulating virtual infrastructure. As already described, in order to emulate a virtual node v at location ℓ_v, we replicate the virtual node at every device within distance $R_1/4$ of location ℓ_v. As mobile nodes join and leave, and as they enter and leave the relevant region near to ℓ_v, they participate—and cease participating—in the emulation. Thus, the basic goal of the emulation algorithm is to ensure the consistency of the replicas, and this is accomplished via the CHA protocol described in Section 3. There are, however, a series of issues not addressed by the CHA protocol.

The first problem relates to virtual nodes communicating with each other. At the end of an instance, the CHA protocol outputs a history that can be used to compute the message broadcast by the virtual node. This suffices for a system with one virtual node that communicates only with clients. A full-scale virtual infrastructure, however, may consist of *many* virtual nodes, and these virtual nodes may communicate with each other. This leads to the following problem. Imagine that virtual node v_1 sends a message to virtual node v_2, and that these two virtual nodes execute their CHA instances simultaneously. Node v_1 cannot calculate the message that it wants to broadcast until the CHA instance completes and a history is determined; yet, for v_2 to receive the message, it must receive the message from v_1 prior to beginning its CHA instance, so that the message from v_1 can be included in its proposal. In order to avoid this problem, our emulation protocol includes two instances of CHA for each virtual round; a virtual node chooses which instance to participate in depending on whether it is sending a message or listening.

The second problem relates to channel contention among emulation instances. The agreement instances of nearby virtual nodes may interfere with each other, resulting in collisions. Moreover, since each contention manager is designed to reduce contention only within a local region, it is not immediate that the situation stabilizes. As for the previous problem, some care is needed in scheduling the agreement instances of the virtual nodes.

The third problem is related to the dynamic participation of mobile nodes. In Section 3, we allowed nodes to fail by crashing; we did not, however, address the case where new nodes arrive. Thus, in the case of virtual infrastructure, a join protocol is required. In addition, when a virtual node fails, a new node may revive it. Thus, we also introduce a reset protocol that reinitializes the virtual node in its initial state. This dynamic behavior leads to several subtleties associated with the contention manager.

In this section, we attempt to give a brief overview of how to generalize the CHA protocol for implementing virtual infrastructure. The complete details can be found in [19].

4.1 Scheduling the Virtual Nodes

We first define a schedule that can be used to avoid contention among distinct virtual nodes that may want to communicate. One of the advantages provided by virtual infrastructure is that virtual nodes are static, and hence more predictable than real nodes; this makes it easy to calculate a good schedule in advance using a centralized algorithm. The length of the schedule is constant, depending only on the density of the virtual node locations.

Let $schedule[0..s-1]$ be an array in which each entry is a subset of the virtual nodes, and assume that virtual node v is located at location ℓ_v. We say that a virtual node v is *scheduled* in some virtual round r if $v \in schedule[r]$; otherwise, we say that it is *unscheduled*. The schedule is *non-conflicting* if no two "neighboring" virtual nodes are scheduled to broadcast at the same time: $\forall i \in [0, s-1], \forall v, v'$, if $v, v' \in schedule[i]$, then $|\ell_v - \ell_{v'}| > R_1 + 2R_2$. We say that the *schedule* is *complete* if every virtual node is scheduled for exactly one round $r \in [0, s-1]$. Finding a complete, non-conflicting schedule is straightforward, based, say, on a coloring of the neighbor graph.

4.2 Contention Management

Each virtual node has its own "regional" contention manager. That is, virtual node v_ℓ location at location ℓ relies on contention manager C_ℓ to reduce contention in the nearby area. The conditions under which a virtual node remains alive and makes progress are closely tied to the guarantees of the contention manager. Recall that in the CHA protocol, progress is guaranteed only when (eventually) some leader is elected. Analogously, for emulating virtual infrastructure, a round emulation is successful in the sense of making progress only when there is a leader among the virtual node replicas.

Thus, the goal of a regional contention manager is to elect "temporary" leaders, i.e., leaders that remain within distance $R_1/4$ of ℓ for sufficiently long, i.e., $2(s+10)$ rounds. This can be reasonably accomplished when there are correct nodes that reside close to location ℓ; even if the are moving away from ℓ at maximum velocity, they will remain sufficiently close for sufficiently long.

[3]Note that we consider an array index to be of constant size.

Given such a contention manager, our virtual infrastructure emulation can guarantee that each virtual node remains alive as long as there are a sufficient number of correct nodes that reside sufficiently close to ℓ; the virtual node makes progress whenever a temporary leader is successfully chosen by the contention manager.

4.3 Outline of the Emulation Protocol

The virtual infrastructure emulation consists of four parts with a total of eleven phases: (1) the message sub-protocol, in which clients and virtual nodes broadcast their messages; (2) the *scheduled* agreement instance, in which CHAP is executed for scheduled virtual nodes; (3) the *unscheduled* agreement instance, in which CHAP is executed for unscheduled virtual nodes; and (4) the join/reset sub-protocol.

The Message Sub-Protocol. The message sub-protocol consists of two phases: the client and the vn phases. In the client phase, each client broadcasts its message for the virtual round, or remains silent if it has no message for the virtual round.

In the vn phase, one (or more) replicas broadcast on behalf of the virtual node. Each replica decides whether v should broadcast a message by calling the calculate-history function from CHAP. A replica also examines two other factors: (1) Is the virtual node scheduled in the current virtual round? If the virtual node is *not* scheduled, then the replica *always* broadcasts the virtual node's message. This counterintuitive rule captures the idea that if the virtual node itself chooses to ignore its schedule, then the replica should ignore the schedule as well. (2) If the virtual node is scheduled, then is the emulator itself advised by the regional contention manager to be active? If the replica is advised to be active, it proceeds to broadcast the virtual node's message.

The Agreement Instances. The emulation protocol contains two instances of the CHA protocol. The emulator for v participates in the *scheduled agreement instance* for virtual round r if v is scheduled in r and in the *unscheduled agreement instance* otherwise. In both cases, the goal is to agree on which messages v should receive.

In the case of the scheduled agreement instance, an additional outcome is a decision as to whether or not the virtual node should broadcast a message. Emulators for unscheduled nodes listen passively during the scheduled instance to determine whether the scheduled virtual nodes chose to broadcast a message.

In the scheduled instance, it is easy to see that there is no interference between emulations as the schedule ensures that no two nearby virtual nodes are scheduled in the same virtual round. In the unscheduled instance, however, more care is needed.

Specifically, in the ballot phase of the unscheduled instance, nearby nodes may interfere with each other, thus preventing any ballots from ever being received. Thus, the ballot phase is instantiated using $s + 2$ rounds (instead of 1 round, as is the case for every other phase). An emulator for virtual node v broadcasts during a particular slot in the ballot phase if it is selected by the schedule. This prevents contention in the ballot phase. The two veto phases proceed as in the CHA protocol (and are potentially subject to interference from neighboring virtual nodes).

The Join and Reset Sub-Protocol. The last three phases are dedicated to the join/join-ack/reset phases. In the join phase, a new emulator broadcasts a request to join, if the virtual node it is trying to join is scheduled for that virtual round. In the join-ack phase, a node p_j sends a join response including the entire current state (or some digest thereof) under the following conditions: (1) p_j has already completed the join protocol; (2) p_j detects a join request or a collision in the preceding join phase; (3) p_j is activated by the regional contention manager, and (4) the virtual node is scheduled for the current virtual round.

If a node p_j attempts to join and does not receive a response, it may reset the virtual node. The protocol first checks that the virtual node has in fact failed; otherwise, a reset could lead to inconsistency (and unnecessarily state loss). Thus, if a new node p_i detects no response to its join request, it continues listens in the reset phase. Each active emulator p_j broadcasts in the reset phase if it has either received a join request or detected a collision in the join or join-ack phases. If the new node p_i does not receive any messages or collisions in the reset phase, then it can safely reset the virtual node, re-initializing its local state and beginning the emulation anew.

5. OPEN QUESTIONS

Interesting open questions include: (1) developing CHA protocols that tolerate weaker collision detectors and contention managers; (2) further reducing message size and overhead; (3) reducing the cost of state transfer, particularly during joining; (4) reducing dependence on a schedule, and the length of virtual rounds; and (5) tolerating malicious disruption. Ongoing implementation projects include: (1) providing suitable time synchronization, collision detection, and contention management; (2) developing virtual infrastructure for TinyOS motes and handheld PDAs; and (3) using virtual infrastructure for routing, robot coordination, and other applications.

Acknowledgments

We would like to thank Calvin Newport for discussions that led to several key insights, as well as Erik Demaine, Sam Madden, and Jennifer Welch for many helpful comments. We would also like to thank the anonymous referees for their many detailed suggestions that have helped to improve the presentation.

6. REFERENCES

[1] M. Aguilera, S. Frølund, V. Hadzilacos, S. L. Horn, S. Toueg. Abortable and query-abortable objects and their efficient implementation. In *PODC*, 2007.

[2] Lali Barrière, Pierre Fraigniaud and Lata Narayanan. Robust position-based routing in wireless Ad Hoc networks with unstable transmission ranges. In *DIAL-M*, 2001.

[3] M. D. Brown. Air traffic control using virtual stationary automata. Master's thesis, MIT, Sept. 2007.

[4] M. Brown, S. Gilbert, N. A. Lynch, C. Newport, T. Nolte, and M. Spindel. The virtual node layer: A programming abstraction for wireless sensor networks. In *WSNA*, April 2007.

[5] C. Busch, M. Magdon-Ismail, F. Sivrikaya, and B. Yener. Contention-free mac protocols for wireless sensor networks. In *DISC*, Oct. 2004.

[6] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4):398–461, 2002.

[7] G. Chockler, M. Demirbas, S. Gilbert, N. Lynch, C. Newport, and T. Nolte. Reconciling the theory and practice of unreliable wireless broadcast. In *Workshop on Assurance in Distributed Systems and Networks*, 2005.

[8] G. Chockler, M. Demirbas, S. Gilbert, C. Newport, and T. Nolte. Consensus and collision detectors in wireless ad hoc networks. In *PODC*, 2005.

[9] G. Chockler, M. Demirbas, S. Gilbert, N. Lynch, C. Newport, and T. Nolte. Consensus and collision detectors in radio networks. *Distributed Computing*, 21(1):55-84, June 2008.

[10] J. Deng, P. K. Varshney, and Z. J. Haas. A new backoff algorithm for the IEEE 802.11 distributed coordination function. In *Communication Networks and Distributed Systems Modeling and Simulation*, Jan. 2004.

[11] S. Dolev, S. Gilbert, L. Lahiani, N. A. Lynch, and T. Nolte. Virtual stationary automata for mobile networks. In *OPODIS*, 2005.

[12] S. Dolev, S. Gilbert, N. A. Lynch, E. Schiller, A. A. Shvartsman, and Jennifer L. Welch. Virtual mobile nodes for mobile ad hoc networks. In *DISC*, Oct. 2004.

[13] S. Dolev, S. Gilbert, N. A. Lynch, Alex A. Shvartsman, and Jennifer Welch. Geoquorums: Implementing atomic memory in mobile ad hoc networks. In *DISC*, Oct. 2003.

[14] S. Dolev, S. Gilbert, N. A. Lynch, A. A. Shvartsman, and J. Welch. Geoquorums: Implementing atomic memory in mobile ad hoc networks. *Distributed Computing*, 18(2):125–155, Nov. 2005.

[15] S. Dolev, S. Gilbert, E. Schiller, A. A. Shvartsman, and J. L. Welch. Autonomous virtual mobile nodes. In *DIALM-POMC*, 2005.

[16] S. Dolev, L. Lahiani, N. A. Lynch, and T. Nolte. Self-stabilizing mobile node location management and message routing. In *SSS*, Oct. 2005.

[17] R. Droms and C. Newport. Virtual infrastructure routing for mobile ad hoc networks. Manuscript, 2007.

[18] R. Gallager. A perspective on multiaccess channels. *IEEE Trans. Information Theory*, IT-31:124–142, 1985.

[19] S. Gilbert. *Virtual Infrastructure for Wireless Ad Hoc Networks*. PhD thesis, MIT, 2007.

[20] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.

[21] T. Herman and S. Tixeuil. A distributed TDMA slot assignment algorithm for wireless sensor networks. In *Workshop on Algorithmic Aspects of Wireless Sensor Networks*, 2004.

[22] I. Keidar and D. Dolev. Increasing the resilience of atomic commit at no additional cost. In *PODS*, 1995.

[23] I. Keidar and D. Dolev. Broadcast in the face of network partitions: Exploiting group communication for replication in partitionable networks. D. Avresky, ed., *Dependable Network Computing*, chapter 3. Kluwer Academic Publications, 2000.

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

[25] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[26] B. W. Lampson. How to build a highly available system using consensus. In *WDAG*, 1996.

[27] N. A. Lynch, S. Mitra, and T. Nolte. Motion coordination using virtual nodes. In *CDC*, Dec. 2005.

[28] Tal Mizrahi and Yoram Moses. Continuous consensus via common knowledge. In *TARK '05*, 2005.

[29] Tal Mizrahi and Yoram Moses. Long Live Continuous Consensus. In *DISC*, 2007.

[30] Tal Mizrahi and Yoram Moses. Continuous Consensus with Ambiguous Failures. In *ICDCN*, 2008.

[31] K. Nakano and S. Olariu. Uniform leader election protocols in radio networks. In *ICPP*, 2001.

[32] K. Nakano and S. Olariu. A survey on leader election protocols for radio networks. In *I-SPAN*, 2002.

[33] R. Newton, Arvind, and M. Welsh. Building up to macroprogramming: An intermediate language for sensor networks. In *IPSN*, 2005.

[34] Tina Nolte. *Virtual Stationary Timed Automata for Mobile Networks*. PhD thesis, MIT, *To appear:* 2008.

[35] T. Nolte and N. A. Lynch. Self-stabilization and virtual node layer emulations. In *SSS*, Nov. 2007.

[36] T. Nolte and N. A. Lynch. A virtual node-based tracking algorithm for mobile networks. In *ICDCS*, June 2007.

[37] J. Polastre and D. Culler. Versatile low power media access for wireless sensor networks. In *ENSS*, 2004.

[38] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *MobiCom*, Aug. 2000.

[39] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[40] M. Spindel. Simulation and evaluation of the virtual node layer. Master's thesis, MIT, June 2008.

[41] D. Skeen. Nonblocking commit protocols. In *SIGMOD*, pages 133–142, 1981.

[42] D. Skeen. A quorum-based commit protocol. In *Berkeley Workshop*, 1982.

[43] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, 2004.

[44] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Mobile systems, applications, and services*, 2004.

[45] D. E. Willard. Log-logarithmic selection resolution protocols in a multiple access channel. *SIAM Journal of Computing*, 15(2):468–477, 1986.

[46] A. Woo, K. Whitehouse, F. Jiang, J. Polastre, and D. Culler. The shadowing phenomenon: implications of receiving during a collision. *Technical Report UCB//CSD-04-1313, UC Berkeley*, March 2004.

[47] J. Wu. Title pending. PhD thesis, CUNY, *To appear: 2008.*