# RAMBONODES for the Metropolitan *Ad Hoc* Network

Jacob Beal Seth Gilbert

*MIT Computer Science and Artificial Intelligence Lab*
*32 Vassar Street, Cambridge, MA 02139*
{jakebeal,sethg}@mit.edu

## Abstract

*We present an algorithm to store data robustly in a large, geographically distributed network. It depends on localized regions of data storage that move in response to changing conditions. For example, data may migrate away from failures or toward regions of high demand. The* PERSISTENTNODE *algorithm [2] provides this service robustly, but with limited safety guarantees. We use the* RAMBO *framework [10, 15] to transform* PERSISTENTNODE *into* RAMBONODE*, an algorithm that guarantees atomic consistency in exchange for increased cost and decreased liveness. A half-life analysis of* RAMBONODE *shows that it is robust against continuous low-rate failures. Finally, we provide experimental simulations for the algorithm on 2000 nodes, demonstrating how it services requests and examining how it responds to failures.*

## 1. Introduction

Robust storage is a key problem in wireless, *ad hoc* networks: data must be maintained in a *reliable*, *accessible*, and *consistent* manner. Robust atomic memory is a favored solution in distributed algorithms.

The PERSISTENTNODE algorithm [2] implements a virtual mobile node that travels through an *ad hoc* network, servicing read/write memory requests. In PERSISTENTNODE, the data is robust and survives even extreme failure conditions; however atomicity is guaranteed only under specific conditions. In this paper, we augment PERSISTENTNODE using the RAMBO framework [10, 15], trading increased communication cost and decreased liveness for unconditional atomicity: the resulting algorithm is RAMBONODE. We then examine the trade-off between consistency and availability in a large geographically distributed network such as a Metropolitan *Ad Hoc* Network (MAN).

By adapting RAMBONODE for the MAN setting, we are also able to make stronger performance guarantees than prior RAMBO papers. We show that the RAMBONODE algorithm can tolerate continuous, ongoing failures, as long as the rate of failures in any region of the network is not too high.

For a more complete version of this paper, see [4].

## 2. The Metropolitan *Ad Hoc* Network Model

We consider the MAN scenario [19], a large city populated by millions to billions of computational devices, each connected to its neighbors by short-range wireless links.

A MAN consists of an unknown number of partially synchronous nodes embedded in Euclidean space with links to all neighbors within a fixed radius. Messages may be lost or reordered, but not corrupted. Nodes fail by stopping and may be rebooted, losing all state and choosing a new UID.

The MAN setting has several key properties: *Locality*: Communication cost is dominated by number of hops, so geographically local algorithms are significantly cheaper than those that require long-distance communication. *Continuous Failures*: It is unrealistic to talk about a fixed *number* of failures; rather, we consider the *rate* of failure within a geographic area. *Immobility*: If nodes are moved only by humans, then most nodes are immobile most of the time, in a large enough network. *Self-Organization*: Due to large scale and ongoing failures, direct human administration is impractical. These systems must be self-organizing and must adapt robustly to changes in the network topology. *No Infrastructure*: We assume no network services, such as routing, naming, and coordinates, due to the practical difficulties of deploying services at this scale.

Other algorithms for the MAN setting includes Beal's prior work with PERSISTENTNODE [2, 3], where the network is partitioned into clusters that can be grouped together to form a hierarchy suitable for tasks like routing.

The study of networks with properties of a MAN has often fallen under the rubric of "sensor networks" (e.g., [6,

*The first author is partially supported by DARPA ONR N00014-02-1-0721. The second author is partially supported by NSF ITR 0121277, AFOSR #F49620-00-1-0097, MURI AFOSR #F49620-00-1-0327, DARPA #F33615-01-C-1896, and NSF 64961-CS.

11, 17]). There are a number of aspects that differentiate the MAN from traditional sensor networks.

Much of the research on sensor networks is organized around the collection and propagation of sensor data. Consider, for example, a typical application, the TinyDB project [16], that has implemented a real-time database that stores consistent data. The database allows a specially designated "root" node to access distributed sensor data, by issuing complex queries. In our model, there is no special root node, and any node can access the shared memory. In general, we aim to enable a MAN to support higher level distributed computation, not just to collect and process data from real-time sensors.

Another aspect typical of sensor networks research is the severe resource constraints imposed by the tiny, lightweight sensor devices: the tiny motes have small batteries and small processors. MAN nodes are less limited: they are not necessarily small, and may be connected to power sources.

Communication bandwidth is still a limiting resource: with billions of nodes participating in the algorithm, the volume of data transmissions must be small.

## 3. Background

In an earlier memo [2], Beal develops PERSISTENTNODE, an algorithm for geographically-optimized atomic memory in the MAN setting. In PERSISTENTNODE, a cluster of nodes maintains replicas of the atomic data. This cluster acts as a "virtual mobile node", moving through the network. The PERSISTENTNODE moves by occasionally choosing a new set of replicas (often including many of the current replicas) and sending the data to these new replicas. By carefully choosing the new replicas, PERSISTENTNODE is able to avoid failed regions of the network, keeping the data near to nodes performing read and write operations.

While the PERSISTENTNODE algorithm implements an atomic shared memory, data consistency is timing dependent: if too many messages between nodes are delayed or lost, atomic consistency can be violated. Our goal, then, is to guarantee atomic consistency, regardless of whether the network is delivering messages rapidly or reliably.

We transform the PERSISTENTNODE algorithm using the RAMBO framework (**R**econfigurable **A**tomic **M**emory for **B**asic **O**bjects), developed by Lynch, Shvartsman, and Gilbert [10, 15] to implement atomic memory in highly dynamic networks. RAMBO guarantees atomicity in all executions.

The RAMBO algorithm is presented in an abstract form, and does not specify what configurations (i.e., quorums of replicas) should be used; nor does it specify when to initiate reconfiguration. Also, RAMBO assumes an all-to-all communication network, and therefore does not operate well in the MAN setting.

The RAMBO algorithm uses replicas to provide fault tolerance. In order to ensure consistency among replicas, each write operation is assigned a unique tag, and these tags are then used to determine which value is most recent. The RAMBO algorithm uses *configurations* to maintain consistency. Each configuration consists of a set of participants and a set of *quorums*, where each pair of quorums intersect. Quorums were first used to solve the problem of consistency in a replicated data system by Gifford [8] and Thomas [20], and later by many others (e.g., Attiya, Bar-Noy and Dolev [1]).

The RAMBO algorithm allows the set of replicas to change dynamically, allowing the system to respond to failures and other network events. The algorithm supports a reconfiguration operation that chooses a new set of participants and a new set of replica quorums. Earlier algorithms also address the reconfiguration problem (e.g. [5, 7, 12, 18]), but RAMBO provides more flexibility, and is therefore more suitable for our application. For more details comparing these algorithms, see RAMBO [10].

Each node maintains a set of active configurations. When a new configuration is chosen, it is added to the set of active configurations; when a configuration is upgraded, old configurations can be removed from the set of active configurations.

RAMBO decouples the reconfiguration mechanism and the read/write mechanism: a separate service is used to generate and agree on new configurations, and the read/write mechanism uses all active configurations. In order to determine an ordering on configurations, a separate consensus service such as Paxos [13] is used.

The use of RAMBO improves on the PERSISTENTNODE algorithm by guaranteeing consistency, while maintaining the ability to tolerate significant and recurring failures. On the other hand, the new algorithm is more expensive, requiring more state and communication, and provides reduced availability: the PERSISTENTNODE algorithm can return a response if even one replica remains active and timely. It is impossible to guarantee a consistent, available, partition-tolerant atomic memory [9]; thus we explore the trade-off between consistency and availability in the MAN setting.

## 4. RAMBONODES

The RAMBONODE algorithm consists of the RAMBO read/write mechanism, the PERSISTENTNODE configuration service, and the Paxos consensus service. (See Figure 1 for an overview of the algorithm.)

**Communication.** In RAMBO, each round depends on gossip-based, all-to-all communication: a round completes when an initiating node learns that a majority of nodes have received gossip messages for that round. The MAN setting

**Write Operations:** (at node $i$)

1. Gossip until $i$ gets tag/value from a majority of nodes in active configurations.
2. Choose new tag.
3. Gossip until $i$ gets acknowledgments that a majority of nodes in active configurations have received new tag/value.

**Read operations:** (at node $i$)

1. Gossip until $i$ gets tag/value from a majority of nodes in active configurations.
2. Gossip until $i$ gets acknowledgments that a majority of nodes in active configurations have received tag/value.

**Reconfiguration:**

1. Node $i$ is designated to initiate a reconfiguration.
2. Initiate broadcast/convergecast to choose new members.
3. Initiate Paxos consensus to agree on the new configuration.
4. Add configuration outputted by Paxos to the set of active configs.

**Configuration Upgrade:** (if there is more than one active configuration)

1. Gossip until $i$ gets a tag/value from a majority of nodes in all old configs.
2. Note largest tag/value.
3. Gossip until $i$ gets acknowledgments that a majority of nodes in the new configuration have received the tag/value.
4. Mark old configuration as removed.

**Figure 1. Overview of RAMBONODE algorithm.**

is conducive to gossip, but it must use only local communication rather than all-to-all. We implement a local gossip service which flows through all active participants, plus all other nodes within $k$ hops, allowing communication across small gaps between active participants.

**Reconfiguration.** Each configuration consists of a cluster of nodes maintaining replicas of the atomic data. The cluster consists of nodes within $P$ hops of the *center*: the node which initiated the configuration. Later, when analyzing the performance of the algorithm, for the sake of simplicity we assume a bound on the maximum density of the network in order to limit the number of active participants. It is easy to develop alternate mechanisms to limit the number of participants (e.g., decreasing $P$ during times of high density).

Every so often, a reconfiguration occurs, choosing a new center and a new set of participants. There are two ways in which potential new centers are chosen.

When the center does not fail, it chooses one of its neighbors to be the new center. This choice is based on an arbitrary distributed heuristic function calculated by gossip among the members of the configuration (as in PERSISTENTNODE). This function biases the direction in which the data moves; for example, the function may attempt to choose a direction in which fewer nodes have failed or from which more nodes send read and write requests.

On the other hand, if a node believes that the center may have failed, then it begins the process of creating a new configuration. Based on available heuristic information, it chooses one of its neighbors to try to become the new center.

A node chosen to become the center runs a broadcast/convergecast to generate a proposal for a new configuration.

When only the center designates a new center candidate, only one node attempts to start a new configuration. In the case where there are failures, many nodes may attempt to become the center of the new configuration. Either way, it is guaranteed that periodically at least one node attempts to start a new configuration.

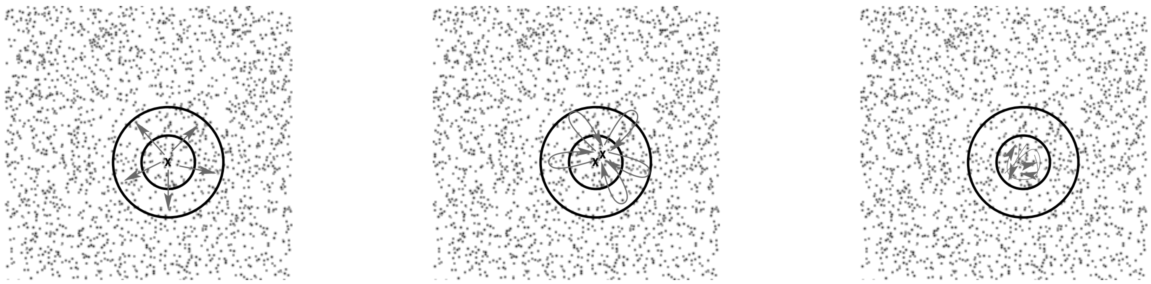
This mechanism implements an eventual leader-election service sufficient to guarantee the liveness of the Paxos consensus algorithm. Each prospective configuration is then submitted to the Paxos consensus service, which ensures that only one of the potentially many prospective leaders succeeds.

The Paxos protocol involves two rounds of gossip in order to agree on a new configuration: in the first, a majority of the old configuration is told to prepare for consensus; in the second, a majority of the old configuration is required to vote on the new configuration. (See [13] for more details.) Then the new configuration is added to the list of active configurations; this information spreads through gossip to members of the old and new configurations.

**Configuration Upgrade.** In order to remove old configurations from the set of active configurations, an *upgrade* operation occurs that upgrades the new configuration, transferring information from the old configurations to the new configuration. The upgrade requires two phases. In the first phase, a node gossips to ensure that it has a recent tag and value. When it has contacted a majority of the nodes in every old configuration, the second phase begins. In the second phase, the node ensures that a majority of nodes in the new configuration receive the recent tag and value. When a majority of nodes in the new configuration have acknowledged receiving the tag and value, the upgrade is complete and the old configurations can be removed. The removal information spreads through gossip to all participants.

**Read/Write Operations.** Each read or write operation consists of two phases. In each phase, the node initiating the operation communicates with majorities for all active configurations. We assume that every node initiating a read or write operation is near some member of an active configuration. If this is not the case, some alternate routing system is used to direct messages to a node that is nearby, which can then perform the read/write operation: in the MAN setting, we focus on local solutions to problems.

We first consider a write operation. In the first phase of the operation, the initiator attempts to determine a new unique tag. The initiating node begins gossiping, collecting tags and values from members of active configurations. When the initiator has received tags and values from a majority of nodes in every active configuration, the first phase is complete. The node then chooses a new, unique tag larger than any tag discovered in the first phase. At this point, the second phase begins. The initiating node begins gossiping the new tag and value. When it has received acknowledg-



(a) The centermost node (shown by an $x$) sends out a poll requesting "goodness" estimates, which determine the new center candidate.

(b) The new center candidate runs a convergecast to discover the members of the new configuration.

(c) The new center begins Paxos to decide on the new configuration. If consensus succeeds, the new configuration is installed.

**Figure 2. Illustration of the reconfiguration process.**

ments from a majority of nodes from every active configuration, the operation is complete.

A read operation is very similar to a write operation. The first phase again contacts a majority of nodes from each active configuration, and thus learns the most recent tag and value. The second phase is equivalent to the second phase of a write operation: the discovered value is propagated to a majority of nodes from every active configuration. This second phase is necessary to help earlier write operations to complete; if the initiator of an earlier write operation fails or is delayed, the later read operation is required to help it complete. This is necessary to ensure atomic consistency.

## 5. Atomic Consistency

The RAMBONODE algorithm guarantees atomic consistency in all executions, regardless of the number of failures, messages lost or delayed, or other asynchronous behavior. The proof closely follows that presented in [10]. For the full proofs, see [4].

## 6. Conditional Performance Analysis

As long as the rate of failure is not too high, read and write operations always complete rapidly. For a more complete analysis, see [4].

**Assumptions.** Good performance of our algorithm depends on four additional assumptions about the failure patterns and the network.

*Half-Failure* assumes that the rate of failure in any region of the network is not too high. If too many nodes in one region can fail, then no localized algorithm can hope to succeed. We say that a timed execution satisfies *$(P, H)$-Half-Failure* if for all balls of radius $P$, for every interval of time of length $H$, fewer than half the nodes in the ball fail during the interval. (The Half-Failure assumption is a generalization of the bounded half-life criteria, introduced by Karger, Balakrishnan, and Liben-Nowell [14].)

The *Partition-Freedom* assumption ensures that nearby nodes are able to communicate with each other. If a partition occurs in the network, our algorithm continues to guarantee consistency; however it is impossible to guarantee fast read and write operations. We say that an execution guarantees *$(P, k)$-Partition-Freedom* if for nodes $i$ and $j$ within $2P$ distance units of each other, there is always a route from $i$ to $j$ of length less than $4kP$ hops, for some fixed $k$.

Assume that $i$ and $j$ are two nodes in the network, and that the distance from $i$ to $j$ is less than the communication radius $r$. We assume that every message sent by $i$ to $j$ is received within time $d$.

Lastly, we assume that nodes are not too densely distributed anywhere on the network. We say that a network is *$(P, N)$-Dense* if for every ball of radius $P$ in the network, there are no more than $2N$ nodes in the ball. (In practice, it is always possible to choose $P$ smaller to reduce $N$, or to direct excess nodes to sleep and save energy.)

**Liveness Analysis.** Choose $\delta = 4Pd$, the time in which any two nodes in a configuration of radius $P$ can communicate. As in all quorum-based algorithms, liveness depends on a quorum (i.e., a majority) of the nodes in active configurations remaining alive. We first notice that our main goal is to ensure that enough nodes in each configuration remain alive for the operations to complete:

**Lemma 6.1** *If $\pi$ is a read or write operation, and throughout the duration of $\pi$ a majority of nodes in each active configuration do not fail, then $\pi$ terminates in $8 \cdot \delta$.*

In order to show that a majority of nodes in a configuration remain alive, we need to determine how long it takes for a new configuration to be fully installed. The key part of this proof is showing that Paxos terminates quickly, outputting a new configuration. If the half-life, $H$, of the algorithm is

| Configuration Radius | Time per Read/Write | Worst Case Read/Write | Time per Recon |
|---|---|---|---|
| **R-Node – 2 hops** | 7.91 | 64 | 81.2 |
| **R-Node – 3 hops** | 11.59 | 96 | 113.5 |
| **R-Node – 4 hops** | 16.45 | 128 | 149.3 |
| **P-Node – 2 hops** | 6 | 6 | 26 |
| **P-Node – 3 hops** | 9 | 9 | 34 |
| **P-Node – 4 hops** | 12 | 12 | 42 |

**Figure 3. Comparison of RAMBONODE and PERSISTENTNODE for node radius 2-4 (in units of maximum message latency).**

large enough, Paxos can terminate.

**Lemma 6.2** *If $H > (40+22\cdot N)\cdot\delta$, then Paxos will output a decision within time $11\cdot\delta\cdot N$.*

This result guarantees that a majority of nodes in each active configuration do not fail. We can then combine Lemmas 6.1 and 6.2:

**Theorem 6.3** *Assume $H > (40+22\cdot N)\cdot 4\cdot P\cdot d$. Then every read and write operation initiated at a participating, non-failing node completes within time $8\delta = 32\cdot P\cdot d$.*

**Discussion.** To put the numbers in perspective, imagine an *ad hoc* sensor network in which nodes are deployed with a density of ten units per square meter. (For example, imagine a smart dust application.) Choose a radius of six meters for a configuration, and assume that adjacent nodes can communicate in one millisecond. Then Theorem 6.3 requires that no more than 50 units fail every five minutes. Except in a catastrophic scenario, this rate of failure is extreme. In smaller configurations, or lower density environments, it becomes even easier to satisfy the Half-Failure property. Figure 4 graphs the permitted failure rates.

## 7. Experimental Results

We simulated the RAMBONODE algorithm, confirming that every execution of RAMBONODE is atomic. We measured the effect of diameter and error rate on performance, demonstrating robustness, but also illustrating the expenses incurred by guaranteeing consistency.

We ran the algorithm in a partially synchronous, event-based simulator with 2000 nodes distributed randomly on a unit square. Communication channels link all nodes within 0.04 units of one another, yielding a network graph approximately 40 hops in diameter. At the beginning of a run, a random node is selected to create the initial configuration. During the experiment, nodes involved in an active configuration randomly invoke read and write operations. Failures are simulated by deleting a node and replacing it with a new one — this happens to any given node in any given round with probability $p_k$.

**Experimental Latencies.** Figure 3 shows operation latency in failure free executions on 2000 nodes. (Simulations with small rates of failure were quite similar.) For configurations with radius $P$, read and write operations take, on average, time $4P$, due to the two phase operations.

The worst case latency for read and write operations in RAMBONODE is significantly worse than the experimental latency. Failures can significantly increase the time an operation takes: the Partition-Failure assumption allows failures to double the cost of communication. Also, inopportune reconfiguration can increase read and write latencies: a new configuration can cause a phase to (effectively) restart.

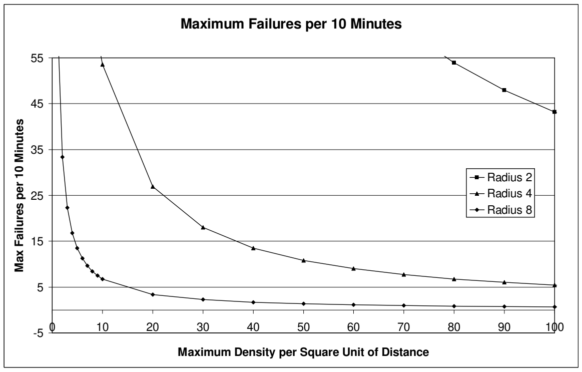
Choosing a new configuration requires more phases than a read or write operation, and thus is slower.

Comparing RAMBONODE to PERSISTENTNODE illustrates the cost of atomic consistency in a MAN. Read and write operations take longer, and reconfiguration is much slower due to the need for consensus. Fortunately, RAMBO decouples read and write operations from reconfiguration.

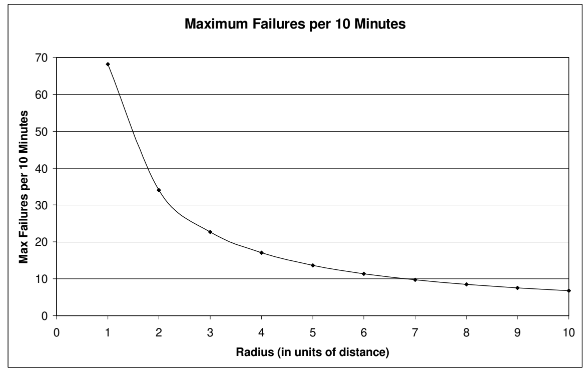
**Configuration Size.** As the radius of a configuration increases, the time to execute an operation and the time to reconfigure are expected to increase linearly in failure-free executions. The data we obtained for configurations of radius two, three, and four suggests that this is the case.

In order to complete an operation or a reconfiguration, RAMBO requires the initiator to collect information on a majority of members of the configuration. This, in turn, requires every node in a configuration to maintain information about the other nodes in the configuration and the ongoing operations. A naive gossip implementation leads to large amounts of storage ($O(N^2)$ per node), which in turn causes the simulation to become untenable for large radius ($P \geq 5$). An improved implementation would reduce the storage (to $O(N)$ per node); nevertheless, note that the RAMBONODE algorithm is only efficient when $P$, the node radius, is relatively small, such that a configuration does not contains too large a number of replicas, i.e. $N$ is not too large. PERSISTENTNODE, by contrast, requires only $O(1)$ storage per node.

**Node Failures.** We ran simulations with varying rates of failure: $p_k$ varied from zero to an expected 20% failure during a single reconfiguration (based on average reconfiguration times). As long as no more than half the nodes failed during a single half-life, the algorithm continued indefinitely to respond to read and write requests, as predicted by Theorem 6.3. We expected to find a sharp transition from 100% success to complete failure of read and write operations, and were not disappointed. From 0-2% failure rate (per expected reconfiguration time), radius three RAMBONODES showed no significant change in time per operation, or number of operations completed. Above 10% failure rate, nodes generally died after a few reconfigurations; the exact behavior varies.




(a) Constant minimum density: all regions remain populated by at least a few nodes at all times.




(b) Minimum density equals 1/100 maximum density. In this case, the maximum density has little effect on the allowable rate of failure; the radius is the key parameter.

**Figure 4. Theoretical maximum rate of failure that the RAMBONODE algorithm can tolerate, when each node communicates with its neighbors once per millisecond.**

## 8. Conclusion

We have combined PERSISTENTNODE and RAMBO, producing RAMBONODE, which captures the safety properties of RAMBO and the locality and mobility properties of PERSISTENTNODE. We have shown that the new algorithm guarantees atomic consistency in all executions, and that the algorithm performs well, as long as the rate of failure is not too high. RAMBONODE is especially suitable for deployment in *ad hoc* networks, like a MAN; it is highly localized, tolerates continuous failures, and requires no network infrastructure.

## References


[1] H. Attiya, A. Bar-Noy, D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[2] J. Beal. Persistent nodes for reliable memory in geographically local networks. Tech Report AIM-2003-11, MIT, 2003.

[3] J. Beal. A robust amorphous hierarchy from persistent nodes. In *CSN*, 2003.

[4] J. Beal, S. Gilbert. RamboNodes for the metropolitan *ad hoc* network. Tech Report AIM-2003-027, MIT, 2003.

[5] K. Birman, T. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP*, 1987.

[6] M. Demirbas, A. Arora, M. Gouda. A pursuer-evader game for sensor networks. In *SSS*, 2003.

[7] A. El Abbadi, D. Skeen, F. Cristian. An efficient fault-tolerant protocol for replicated data management. In *PODS*, 1985.

[8] D. K. Gifford. Weighted voting for replicated data. In *SOSP*, 1979.

[9] S. Gilbert, N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Sigact News*, 2001.

[10] S. Gilbert, N. Lynch, A. Shvartsman. RAMBO II:: Rapidly reconfigurable atomic memory for dynamic networks. In *DSN*, 2003.

[11] J. Hill, R. Szewcyk, A. Woo, D. Culler, S. Hollar, K. Pister. System architecture directions for networked sensors. In *ASPLOS*, 2000.

[12] S. Jajodia, D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *Trans. on Database Systems*, 15(2):230–280, 1990.

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

[14] D. Liben-Nowell, H. Balakrishnan, D. Karger. Analysis of the evolution of peer-to-peer systems. In *PODC*, 2002.

[15] N. Lynch, A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *DISC*, 2002.

[16] S. R. Madden, R. Szewczyk, M. J. Franklin, D. Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Workshop on Mobile Computing and Systems Applications*, 2002.

[17] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler. Wireless sensor networks for habitat monitoring. In *2002 ACM International Workshop on Wireless Sensor Networks and Applications*, 2002.

[18] R. D. Prisco, A. Fekete, N. A. Lynch, A. A. Shvartsman. A dynamic primary configuration group communication service. In *DISC*, 1999.

[19] T. Shepard. Personal Communication.

[20] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *Transactions on Database Systems*, 4(2):180–209, 1979.