Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2003-034
AIM-2003-027

December 17, 2003

RamboNodes for the Metropolitan Ad Hoc Network

Jacob Beal and Seth Gilbert

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Abstract

We present an algorithm to store data robustly in a large, geographically distributed network by means of localized regions of data storage that move in response to changing conditions. For example, data might 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[17, 11] to transform PERSISTENTNODE into RAMBONODE, an algorithm that guarantees atomic consistency in exchange for increased cost and decreased liveness. In addition, 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

As wireless devices proliferate and multiply in numbers, it will become increasingly impractical to administer them on a per-device level. An attractive alternative is to have the devices self-organize into *ad hoc* computing platforms.

A key problem that must be addressed for any such platform is data storage. If a user is to entrust her data to a distributed system, she must be able to trust that her data will not be lost, will remain readily accessible, and will be consistent. This is not only a user-level issue: robust, atomic memory is a fundamental building block of many distributed algorithms and facilitates the construction of higher level services.

The PERSISTENTNODE algorithm[2] implements a virtual mobile node that travels through the static network, servicing read/write memory requests. Each PERSISTENTNODE is a key/value pair replicated on a set of topologically close members of the network. Once created, a PERSISTENTNODE engages in self-repair and migrates through the network in response to changing conditions — for example, a PERSISTENTNODE may be programmed to avoid regions where failures have occurred, or to move toward regions where its data is in demand. PERSISTENTNODE is designed with large, geographically distributed networks (e.g., a Metropolitan *Ad Hoc* Network) in mind. It operates correctly, however, in general networks, albeit poorly in networks of low diameter.

The problem with PERSISTENTNODE is that while the data is robust and remains live under extreme failure conditions, the atomicity guarantee holds only under certain, specific conditions. In this paper, we augment PERSISTENTNODE using the RAMBO framework[17, 11], trading increased communication cost

*The first author is partially supported by DARPA grant ONR N00014-02-1-0721. The second author is partially supported by NSF ITR Grant 0121277, AFOSR Contract #F49620-00-1-0097, DARPA Contract #F33615-01-C-1896, NSF Grant 64961-CS, and NTT Grant MIT9904-12.

and decreased liveness for unconditional atomicity, and calling the resulting algorithm RAMBONODE. An important contribution of this work, then, is examining the trade-off between consistency and availability in the MAN setting.

The RAMBONODE algorithm is also the first implementation of a RAMBO-based algorithm which includes all reconfiguration details. Prior work on RAMBO left out several practical components of the algorithm necessary for a real implementation of atomic memory. For example, it was assumed that an external service determined who the current owners of the data should be and when reconfigurations should occur. As a result, in devising RAMBONODE, we answered a number of open questions presented in [11] regarding the practical implementation of a RAMBO algorithm.

As a result, we are able to make stronger performance guarantees than are possible in the prior RAMBO papers. In particular, we focus on showing 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. This type of failure analysis, while immensely useful in understanding the real-world applications of an algorithm, is relatively rare among formally analyzed algorithms, and therefore is an important property of our new algorithm.

In this paper, then, we present and analyze the RAMBONODE algorithm. First, in Section 2, we provide a brief overview of the MAN setting and the general network model. In Section 3, we summarize the PERSISTENTNODE algorithm and the RAMBO algorithm. In Section 4 we describe the RAMBONODE algorithm in more detail, and in Sections 5 and 6 we provide a formal analysis of the algorithm, proving atomicity and conditional performance guarantees. We also compare the theoretical performance of RAMBONODE with PERSISTENTNODE, in order to understand the cost of consistency. We then present experimental results from simulations on 2000 nodes in Section 7, confirming our theoretical results.

2 The MAN System Model

In this section, we first describe the challenges posed by the Metropolitan *Ad Hoc* Network (Section 2.1). We then present, more formally, the network model used in this paper (Section 2.2). Finally, we compare the MAN setting to other related types of networks, and discuss how our work compares to algorithms developed for these networks (Section 2.3).

2.1 The Metropolican *Ad Hoc* Network

In the MAN scenario[22] we consider a large city, populated by millions to billions of computational devices, each connected to its nearby neighbors by short-range wireless links. The MAN setting introduces several difficult challenges, and a key contribution of this paper is an algorithm capable of handling these issues:

- *Locality*: Short-range communication links imply that the network has a high diameter, and thus the cost of communication will be dominated by the number of hops a message must travel. This also means that many tasks, for example, memory coherence, are significantly cheaper for geographically local algorithms.

- *Continuous Failure*: With millions or billions of nodes, it is unrealistic to talk about a certain *number* of failures during execution of the algorithm. Rather, we consider the *rate* of failure within a unit of geographic area.

- *Immobility*: The sheer number of nodes in the network implies that most nodes can be assumed to be immobile most of the time. (We assume that the most nodes are only moved by the actions of humans.)

- *Self-Organization*: Direct administration of a network this large is impractical (to say the least), particularly under assumptions of continuous failure. Accordingly, these systems must be self-organizing, requiring minimal human intervention, and they must adapt robustly to changes in the network topology.

- *No Infrastructure*: We make no dependence on any pre-organized network structure, notably including routing, naming, and coordinate services. While much research has been done on these infrastructural services, it still remains quite challenging, especially in practice, to reliably provide these services in such a difficult environment.

Other usage of the MAN setting includes Beal's prior work with the PERSISTENTNODE algorithm in [2] and in [3], where persistent nodes are used to partition the network into clusters that can be grouped together to form a hierarchical partition of the network suitable for tasks like routing.

2.2 Formal Details

More formally, the MAN consists of an unknown, though bounded, number of partially synchronous nodes. Nodes may fail by stopping (crashing). A node that has failed may restart, as long as it chooses a new unique process identifier (and reinitializes its state).

The nodes are placed on a two-dimensional plane,[1] and they are connected by a network that allows any node to send a message to any other nearby node (i.e., any node within a fixed, small distance). For the purposes of correctness, no assumption is made as to how long a message takes to be delivered, or whether it is lost in transit.

2.3 Relation to Sensor Networks

The study of networks with some of the properties of a Metropolitan *Ad Hoc* Network has often fallen under the rubric of "sensor networks" (e.g., [12, 20, 5]). There are a number of aspects that differentiate the Metropolitan *Ad Hoc* Network setting from traditional sensor network applications. While the results described in this paper are equally applicable in more typical sensor networks, we were motivated by the example of the MAN, and the PERSISTENTNODE and RAMBONODE algorithms were designed with that in mind.

Much of the research on sensor networks is organized around the collection and propagation of sensor data. Consider, for example, a typical sensor network application, the TinyDB project[19], that has implemented a real-time database that stores consistent data. The database allows a special 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 want 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. In contrast, nodes in the MAN environment are not necessarily as resource limited: MAN nodes are not necessarily small, and may be connected to power sources.

As in sensor networks, however, communication bandwidth is a limited resource. With billions of nodes all attempting to participate in the algorithm, it is important to limit the amount of data transmitted between

[1] Note that this is significantly stronger than the assumptions necessary for the results presented in this paper.

nodes.

3 Background

In this section, we discuss two prior algorithms, PERSISTENTNODE and RAMBO. Components of both of these algorithms play an important role in the new algorithm, and it is therefore useful to review these algorithms.

PERSISTENTNODE In an earlier memo [2], Beal began considering the problem of geographically-optimized atomic memory in the MAN setting, and developed the PERSISTENTNODE algorithm. In this algorithm, a cluster of nodes is designated to maintain replicas of the atomic data. This cluster can be thought of as a "virtual mobile node" that travels around the network according to certain movement rules. The PERSISTENTNODE moves by occasionally choosing a new set of nearby replicas (generally almost identical to the current set) and sending the data to these new nodes. By choosing the new nodes according to its movement rules, the PERSISTENTNODE is able to maintain the data far away from failed regions of the network and near to where the data is needed.

The PN algorithm is the direct predecessor of the algorithm presented in this paper. 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 is no longer ensured. Our goal, then, is to guarantee atomic consistency, regardless of whether the network is delivering messages rapidly or reliably.

RAMBO We transform the PERSISTENTNODE algorithm using the RAMBO framework (**R**econfigurable **A**tomic **M**emory for **B**asic **O**bjects), an algorithm developed by Lynch, Shvartsman, and Gilbert to provide atomic memory in highly dynamic networks.[17, 11] They show that RAMBO guarantees atomic consistency in all executions, regardless of all forms of asynchrony and failures: delayed messages, lost messages, failed nodes, etc.

The RAMBO algorithm, however, is presented in a fairly abstract form, and is missing many components needed for a practical implementation. In particular, it 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. A contribution of this work, then, is the design of an algorithm using the RAMBO framework that is adapted to 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 [23] and much research has followed from this. Of particular note, Attiya, Bar-Noy and Dolev [1] use majorities of processors to implement consistent atomic memory, and their approach is extended by the RAMBO algorithm.

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. Other earlier algorithms also address the reconfiguration problem (e.g. [6, 13, 21, 7, 4]), but RAMBO provides more flexibility, and is therefore more suitable for our application. In particular, it decouples the read and write operations from the reconfiguration process, so that even during a long reconfiguration, read and write operations can complete rapidly. For more details comparing these algorithms, see the full RAMBO paper [18].

Each node maintains a set of active configurations. When a new configuration is chosen, it is added to the set of active configurations, and when a configuration upgrade operation occurs (upgrading the most recent active configuration), old configurations can be removed from the set of active configurations.

When a node wants to perform a read or a write operation, it performs a two phase operation. In each phase, the node communicates with one quorum from each active configuration. Since all pairs of quorums in a configuration intersect, this ensures atomic consistency.

An important feature of RAMBO is the decoupling of 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, implemented using Paxos [14], for example, is used.

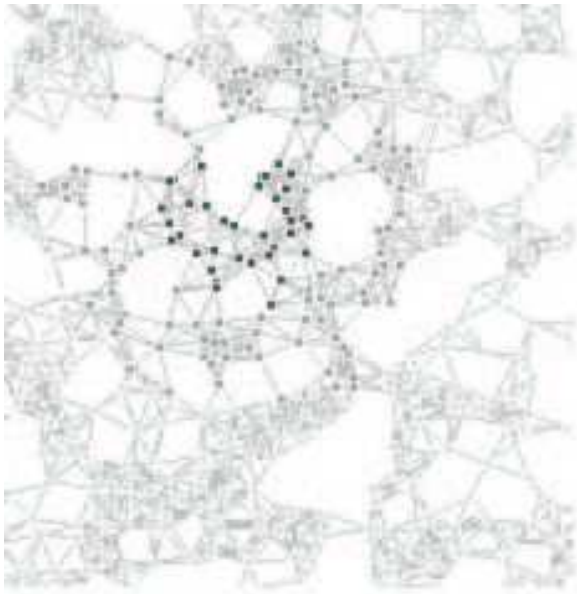


Figure 1: Screenshot from a simulation of RAMBONODE on a 2000 node MAN. Black nodes at the center are in the configuration. Medium gray nodes surrounding the center participate in gossip communication, but are not members of the configuration. Light gray nodes are non-participating. Light gray lines indicate communication links.

RAMBONODE VS. PERSISTENTNODE 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 significantly more state and communication, and provides reduced availability: the PERSISTENTNODE algorithm can return a response if even one replica remains active and timely. Since it is impossible to guarantee a consistent, available, partition-tolerant atomic memory [10], this work allows us to explore the trade-off between consistency and availability in the MAN setting: the new algorithm guarantees consistency, but is only available when the network is well connected; the PERSISTENTNODE algorithm guarantees availability at all times, but only guarantees consistency when the network is well connected.

4 RAMBONODES

The RAMBONODE algorithm consists of a PERSISTENTNODE configuration service combined with the read/write mechanism in RAMBO and the Paxos consensus esrvice. See Figure 2 for a high level description of the algorithm. We first discuss some general issues related to communication, then we present the RAMBONODE reconfiguration algorithm, and finally present the read/write algorithm.

Read/Write Operations:

For a write operation at node i:

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

For read operation at node i:

1. Gossip until node i receives tag/value from a majority of nodes in active configurations.
2. Begin second phase.
3. Gossip until i receives acknowledgments that a majority of nodes in active configurations have received tag/value.

Reconfiguration:

If node i is the center, or a neighbor of i fails:

1. Designate a neighbor to initiate a reconfiguration.

If a neighbor designates i to initiate a reconfiguration:

1. Initiate broadcast/convergecast to choose new members.
2. Initiate Paxos consensus to agree on the new configuration.
3. Add configuration outputted by Paxos to the set of active configurations.

Configuration Upgrade:

If there is more than one active configuration:

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

Figure 2: High level description of the RAMBONODE algorithm for node i.

4.1 Communication

RAMBO as previously specified depends on gossip-based, all-to-all communication: every so often, every node sends portions of its state to every other node. The gossip based nature of RAMBO makes it conducive to the MAN setting; however the algorithm must be adapted to require only local communication, rather than all-to-all communication. To this end, we implement a local communication gossip service. The local gossip flows through all active participants, plus all other nodes within k hops of an active participant, allowing communication across small gaps between active participants.

Much of the algorithm proceeds in phases. In each phase, the initiating node begins gossiping with its neighbors. When it learns that a majority of nodes have received gossip messages from that phase, then the

phase is complete.

4.2 Reconfiguration

At any given time, there exists a tight cluster of nodes maintaining replicas of the atomic data. Therefore, the active participants in our algorithm are a set of nodes within a radius, P, of the last node to successfully complete a reconfiguration. We refer to the node that initiated the last reconfiguration as the *center* of 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. Alternate mechanisms to limit the number of participants (such as decreasing P during times of high density) could easily be developed.

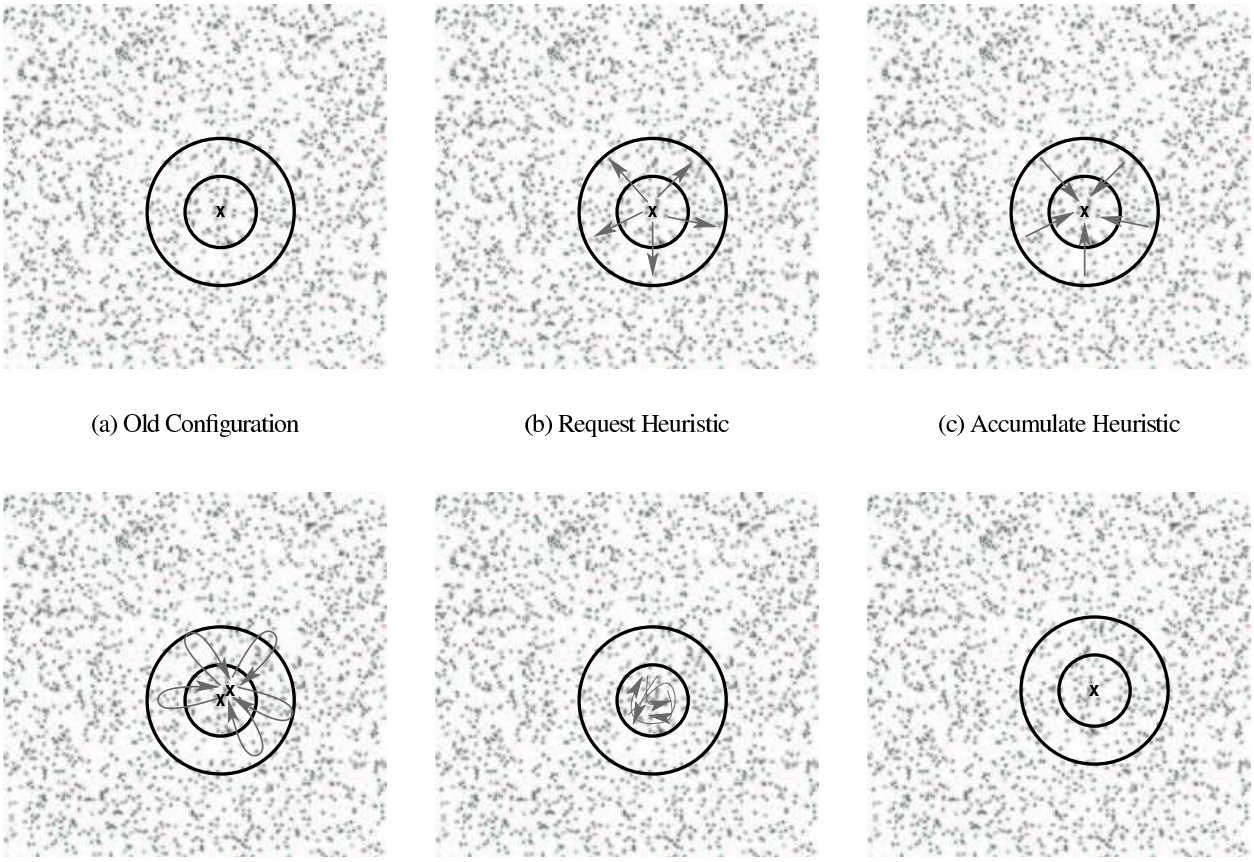
Every so often, a reconfiguration occurs, choosing a new center and a new set of participants. In the normal case, when the center does not fail and the messages sent by the center are delivered, the center chooses one of its neighbors to be the new center, based on an arbitrary distributed heuristic function calculated by gossip among the members of the configuration (as in the PERSISTENTNODE algorithm). This function may be used to bias 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. The chosen neighbor then runs a broadcast/convergecast to generate a proposal for a new configuration.

On the other hand, if failures occur and some arbitrary node in the current set of participants notices that a neighbor has failed (in particular, if a parent in the spanning tree rooted at the center fails), then the node anoints one of its neighbors to try to become the new center. This chosen neighbor then also runs a broadcast/convergecast to generate a proposal for a new configuration.

As a result, in the common case, 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 at least one node attempts to start a new configuration.

This mechanism essentially 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 (i.e., two phases) in order to agree: in the first, a majority of the old configuration is told to prepare for consensus; in the second, a majority of the old



(a) Old Configuration (b) Request Heuristic (c) Accumulate Heuristic

(d) Discover Members (e) Gossip Consensus (f) New Configuration

Figure 3: Changes to a new configuration (and location) occur via the following sequence. From the old configuration [a] the centermost node (shown by an x) sends out a poll [b] requesting "goodness" estimates from everything within distance 2r of the node. These are accumulated to the center [c] which uses it to chooses a high-value nearby node as candidate to become the new center. The new center candidate runs a convergecast to discover what nodes will be in the new configuration [d] and gossips for Paxos consensus on the new configuration [e]. If consensus succeeds, then the new configuration is installed [f]. Under failure, this process runs identically, except that it may be multiplied by many nodes believing they are centermost and some processes dying.

configuration is required to vote on the new configuration. (See [14] for more details on Paxos.) When this is complete, the new configuration is added to the list of active configurations; this information spreads through gossip to members of the old and new configurations.

4.3 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 operation 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.

4.4 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. For the moment, 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 acknowledgments 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

In this section, we show that the RAMBONODE algorithm guarantees atomic consistency in all executions. The RAMBONODE algorithm was developed using the RAMBO framework, and therefore the proof of atomic consistency closely follows that presented in [9]. We sketch the important ideas here.

In order to show that an algorithm guarantees atomic consistency, it is sufficient to show that for every execution, α, of the algorithm, there exists a partial ordering, $\prec$, of the read and write operations with the following properties: (i) the partial order, $\prec$, totally orders all write operations in α, (ii) the partial order, $\prec$, orders every read operation in α with respect to every write operation in α, (iii) for each read operation, if there is no preceding write operation in $\prec$, then the read operation returns the initial value; otherwise, the read operation returns the value of the unique write operation immediately preceding it in $\prec$, and (iv) if some operation, π_1, completes before another operation, π_2, begins, then π_2 does not precede π_1 in $\prec$. If π_2 is a write operation, then $\pi_1 \prec \pi_2$. (See Lemma 13.16 in [16].)

In our case, we choose a partial ordering that is consistent with the sequence tag associated with each operation. Properties (i)–(iii) are self-evident; we discuss Property (iv) in further detail.

Lemma 5.1 *Assume π_1 and π_2 are two read or write operations such that π_1 completes before π_2 begins. Then $tag(\pi_1) \leq tag(\pi_2)$, and as a result, $\pi_2 \not\prec \pi_1$; if π_2 is a write operation, then $tag(\pi_1) < tag(\pi_2)$, and as a result, $\pi_1 \prec \pi_2$.*

Proof (sketch). If the sets of active configurations overlap, then this lemma follows from the quorum intersection property. That is, if there is some configuration, c, active both during the second phase of π_2 and the first phase of π_1, then π_1 must have updated a write-quorum of c, and π_2 queried a read-quorum of c. As a result, $tag(\pi_1) \leq tag(\pi_2)$, and if π_2 is a write then the inequality is strict.

Let s_1 be the largest configuration active during the second phase of π_1. Let s_2 be the smallest configuration active during the first phase of π_2. Assume that $s_1 < s_2$. Otherwise, it must be the case that there is

some configuration that was active during both operations.

Then there must exist some sequence of configuration-upgrade operations, $\gamma_1, \gamma_2, \ldots, \gamma_k$ that occur between π_1 and π_2 and remove all the configuration that are smaller than s_2 and $\geq s_1$. (Otherwise, configuration s_1 would still be active when π_2 began.)

We can find such a sequence where $tag(\gamma_1) \leq tag(\gamma_k)$, i.e., each upgrade operation passes on the tag to the next operation. It is also not difficult to show that $tag(\gamma_k) \leq tag(\pi_2)$ (and the inequality is strict if π_2 is a write operation), since s_2 is the smallest active configuration during π_2. Similarly, we realized that $tag(\pi_1) \leq tag(\gamma_1)$.

We therefore conclude that $tag(\pi_1) \leq tag(\gamma_1) \leq tag(\gamma_k) \leq tag(\pi_2)$, and if π_2 is a write operation the inequality is strict. □

Having shown that the partial ordering meets the four requirements for atomicity, we can state the following theorem:

Theorem 5.2 *The* RAMBONODE *algorithm guarantees atomic consistency in all executions, regardless of the number of failures, messages lost or delayed, or other asynchronous behavior.*

6 Conditional Performance Analysis

We next consider the liveness properties of our implementation. In this section, we show that as long as the rate of failure is not too high, read and write operations will always complete rapidly (and the RAMBONODE will not fail). We first present some additional assumptions needed to guarantee liveness (Section 6.1), and then we prove that if these conditions are met, we realize good performance (Section 6.2). Finally, we discuss some of the implications of the theoretical results (Section 6.3).

6.1 Assumptions

Good performance of our algorithm depends on four additional, reasonable, assumptions about the way in which failures occur and the network on which the algorithm runs: (a) *Half Failure*, (b) *Partition Freedom*, (c) *Reliable Message Delivery*, (d) *Maximum Network Density*,

Half-Failure The *Half-Failure* assumption is the most important additional assumption. It requires that the rate of failure in any given part of the network not be too high. If too many nodes in one area can fail, then no localized algorithm can hope to succeed, and an alternate global algorithm must be used to maintain the data.

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 [15]. They require that over a specified half-life, fewer than half the active nodes fail, and no more than twice the nodes join. The smaller the half-life of an algorithm, the higher rate of failure it can tolerate.

We modify their definition by focusing on a localized region of the network (i.e., any ball of radius P), and we ignore the limitations on joining (as that has no negative effect on our algorithm).

Partition-Freedom The *Partition-Freedom* assumption ensures that nearby nodes are really able to communicate efficiently 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 require first that there are no partitions in the network. We also require that the route between any two nearby nodes does not grow too long: if two nearby nodes are forced to communicate through a long chain of intermediate nodes, then it is effectively a "local" partition and read and write operations may be delayed.

We say that a timed execution guarantees *(P, k)-Partition-Freedom* if for all nodes i and j that are within $2P$ distance units of each other, there is always a route from i to j of length less than $4kP$ hops[2].

Reliable Message Delivery 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. (In practice, we can tolerate some messages being lost or delayed, as long as information arrives by some channel within the necessary time bounds.)

Network Density 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

[2]The algorithm is more robust when implemented with k larger; for the analysis we will assume $k = 1$

$2N$ nodes in the ball.

In practice, this is not a particularly severe restriction. It is always possible to choose P smaller, in order to reduce the density. Alternatively, excess nodes could always sleep, saving energy.

6.2 Liveness Analysis

In this section, we present our theoretical performance results, and a few of the ideas in the proof. Details have been omitted due to lack of space.

For the rest of this section, we assume that the requirements of Section 6.1 hold. We show that read and write operations are guaranteed to terminate rapidly.

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. The primary difficulty in ensuring that operations complete rapidly is actually in ensuring that enough nodes remain alive in each configuration for the operations to complete at all:

Lemma 6.1 *If π is a read or write operation, and throughout the duration of π a majority of nodes in each active configuration fail, then π terminates in $8 \cdot \delta$.*

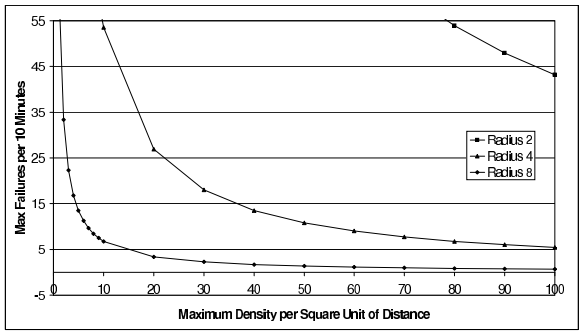
In order to show that a majority of nodes in a configuration remain alive, we need to determine how long it takes for a new configuration to be fully installed. The key part of this proof is showing that Paxos terminates quickly, outputting a new configuration. Notice that Paxos itself will only complete if a majority of nodes in the prior configuration remain alive. Therefore we must assume that the half-life, H, of the algorithm is large enough to allow Paxos to terminate.

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

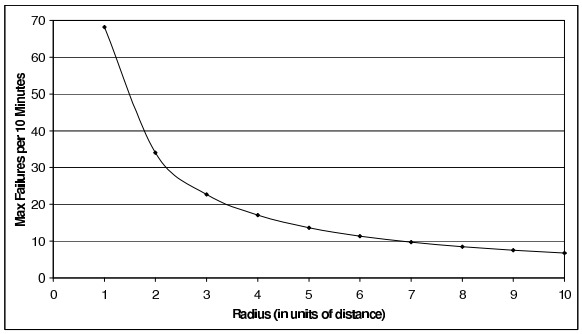
We can then prove the main result by combining Lemmas 6.1 and 6.2:

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

The argument here is similar to that in [18], Theorem 8.17, which shows that as long as the algorithm guarantees enough configuration viability, then read and write operations terminate rapidly.



(a) Constant Minimum Density



(b) Minimum Density Equals 1/100 Maximum Density

Figure 4: Theoretical maximum rate of failure that the RAMBONODE algorithm can tolerate, when each node communicates with its neighbors once per millisecond. Graph (a) assumes a constant minimum density, i.e., that regions remain populated by at least a few nodes at all times. As long as the maximum density is not too large, or the radius of the RAMBONODE is not too big, then a reasonable rate of failures is tolerated. Graph (b) assumes instead that there is a bounded ratio between the minimum density and the maximum density, i.e., that the density stays within a predetermined range. In this case, the maximum density has little effect on the allowable rate of failure. The radius of the RAMBONODE is the important parameter. As long as the radius is not too large, a reasonable rate of failures is tolerated.

6.3 Discussion

In order 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 only that no more than 50 units fail every five minutes. Except in a catastrophic scenario, this rate of failure is extreme. In smaller configurations, it becomes even easier to satisfy the Half-Failure property; in the same system, with configuration of radius one, it is only necessary to ensure that no more than half the units fail in a 1.6s interval. Similarly, decreasing the density only helps reduce the Half-Failure interval, though also decreasing the allowable failure rate. (Figure 4 graphs the permitted failure rates.)

Configuration Radius	Time per Read/Write	Theoretic Worst Case Read/Write	Time per Recon
Rambo			
2 hops	7.91	64	81.2
3 hops	11.59	96	113.5
4 hops	16.45	128	149.3
PersistentNode			
2 hops	6	6	26
3 hops	9	9	34
4 hops	12	12	42

Figure 5: Comparison of RAMBONODE and PERSISTENTNODE latencies, for configurations of varying radius. Average time per operation and average time per reconfiguration in the failure-free case, along with theoretic worst-case latency for read/write and recon operations in failure-prone executions, in units of d, where d is the maximum time for a message to travel between two neighboring nodes, and N is the expected number of nodes in a configuration.

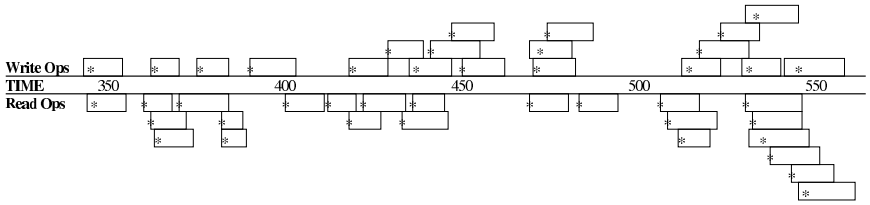


Figure 6: Experiments simulating 2000 particles produce atomic traces of a radius three RAMBONODE. This diagram shows serialization points (asterisks) for 43 operations, excerpted from a 316 operation trace (including events for operations not completely contained in the trace).

7 Experimental Results

We implemented the RAMBONODE algorithm in order to verify that it behaves as predicted.[3] See Figure 1 for a screenshot of a simulation in progress. As expected, every execution of RAMBONODE is atomic. We ran experiments to determine the effect of node diameter and error rate on performance. These experiments demonstrate the robustness of the algorithm, but also illustrate the expenses incurred by guaranteeing consistency.

We ran the implemented algorithm in a partially synchronous, event-based simulator with 2000 particles distributed randomly on a unit square. Communication channels link all particles within 0.04 units of one another, yielding a network graph approximately 40 hops in diameter. At the beginning of a run a random particle is selected to create the initial configuration. During the experiment particles involved in an active

[3]Code written by Jacob Beal is available online at:
http://www.swiss.ai.mit.edu/projects/amorphous/Dynamic/demos/hlsimJ/

configuration randomly invoke *read* and *write* operations. Failures are simulated by deleting a particle and filling its place with a new one — this happens to any given particle in any given round with probability p_k.

We verified the correctness of the algorithm by serializing events into a sequence of atomic operations. As expected, all runs preserved atomicity, even with quite high rates of failures. (Of course, when the failure rates were too high, operations did not complete; however consistency was never violated.) Figure 6 contains a typical fragment of an event trace. This excerpt (from a radius three RAMBONODE) contains 32 events, of which 24 event comprise 12 complete operations and eight are part of operations not completely contained in the timespan considered. The serialization points have been inserted for the entire 337 operation sequence to verify atomicity.

Read and Write Latency In Figure 5, we present data on the latency of various operations. The simulation data in this figure comes from failure free executions of the algorithm on 2000 nodes. (Simulations with small rates of failure were quite similar.) Read and write operations take, on average, time proportional to twice the diameter of the configuration. This is a result of the two phase operations: in each phase, the initiating node must communicate with other nodes that are, on average, half the diameter's distance away; round-trip communication with these nodes takes time proportional to the diameter of the configuration. (See Appendix A for more detailed execution data.)

It is also interesting to note that the worst case latency for read and write operations is significantly worse. This can be attributed to two main factors. First, failures can significantly increase the time an operation takes. The Partition-Failure assumption allows failures to occur in such a way as to double the cost of communication. Also, the pattern of failures may ensure that only the most distant nodes in a configuration remain alive. Second, inopportune reconfiguration can also have some effect on read and write latencies; in particular, if a reconfiguration completes just before an operation completes, the operation must now contact a quorum from the new active configuration.

Reconfiguration Latency Choosing a new configuration requires more phases than a read or write operation, and thus takes significantly longer. Even so, the latency of reconfiguration in the simulation is an order of magnitude faster than the theoretic worst-case latency. This reflects the difference between randomized and adversarial failures. The consensus operation has the potential to take a very long time to complete: each failure can significantly postpone the termination of consensus, requiring the process to essentially

restart. In practice, however, this rarely happens.

Comparison to PERSISTENTNODE One of the goals of this paper is to examine the costs of atomic consistency in a MAN, and therefore we compare RAMBONODE to PERSISTENTNODE, an algorithm that does not guarantee consistency. The first thing to note is that, in general, read and write operations are only slightly slower using RAMBONODE. Worst case read and write times are significantly worse, and this accurately represents the cost of handling the failures in an atomicity preserving manner. Reconfiguration, on the other hand, is significantly more expensive when using RAMBONODE. The need for consensus significantly slows down the reconfiguration operations, and leads to the significantly slower worst case times. These comparisons in some ways justify the design goals of RAMBO: the burden for consistency is placed on the reconfiguration service, allowing read and write operations to continue as usual.

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: each phase of an operation requires communication with a majority of nodes in a configuration. The farther away these nodes are, the longer a phase takes. The data we obtained for configurations of radius two, three, and four suggests that this is, in fact, the case. For configurations with radius greater than four, however, the configurations begin to contain many particles, and the simulation becomes quite slow.

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 particle in a configuration to maintain information about the other particles in the configuration and the ongoing operations. A naive gossip implementation leads to large amounts of storage ($O(N^2)$ per particle), which in turn causes the simulation to become untenable for large simulations when $P \geq 5$. An improved implementation would reduce the storage (to $O(N)$ per particle); nevertheless, it is worth noting that the RAMBONODE algorithm is only efficient when P, the radius of a configuration, is relatively small, and therefore 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 Finally, we ran simulations with varying rates of failure, ranging p_k from zero to an expected 20% failure during a single reconfiguration (based on actual, not worst-case, reconfiguration times).

As long as the failure rate was low enough so that no more than half the nodes failed during a single half-life (i.e., H), 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 behavior of any given test, on the other hand, varies greatly.

8 Conclusion

We have combined the PERSISTENTNODE and RAMBO algorithms to produce the RAMBONODE algorithm which captures the safety properties of RAMBO and the locality and mobility properties of the PERSISTENTNODE algorithm. 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. The RAMBONODE algorithm is especially suitable for deployment in *ad hoc* networks, like a MAN. The algorithm is highly localized, tolerates continuous failures, and requires no network infrastructure.

The MAN setting motivates a number of interesting open problems related to building and understanding the primitive services for distributed computation in a MAN. In this paper, we have examined the cost of atomic consistency in the MAN setting. Are there efficient implementations of other strong primitives, such as atomic broadcast and group communication? Gossip-based communication seems quite promising for the MAN setting, however naive implementations lead to expensive memory requirements for strongly consistent algorithms. What problems can be solved efficiently using gossip-based protocols in a MAN?

There are also open questions related to the current PERSISTENTNODE and RAMBONODE algorithms. The efficiency of the implementation can be improved. There are questions of how we can use these algorithms, in combination with routing and clustering services, to provide higher-level resilience to correlated failures. Finally, it might be interesting to see how these types of algorithms could be used in more traditional sensor network applications. Can these algorithms for persistent data be used to enhance the fault-tolerance of a data-collection network?

References

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

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

[3] Jacob Beal. A robust amorphous hierarchy from persistent nodes. In *to appear in CSN 2003*, 2003.

[4] Ken Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, December 1987.

[5] M. Demirbas, A. Arora, and M. Gouda. A pursuer-evader game for sensor networks. In *Sixth Symposium on Self-Stabilizing Systems*, June 2003.

[6] A. El Abbadi, D. Skeen, and F. Cristian. An efficient fault-tolerant protocol for replicated data management. In *Proc. of the 4th Symp. on Principles of Databases*, pages 215–228. ACM Press, 1985.

[7] B. Englert and A.A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proceedings of the International Conference on Distributed Computer Systems*, pages 454–463, 2000.

[8] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh symposium on operating systems principles*, pages 150–162, 1979.

[9] Seth Gilbert. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. Technical Report LCS-TR-890, M.I.T., 2003.

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

[11] Seth Gilbert, Nancy Lynch, and Alex Shvartsman. RAMBO II:: Rapidly reconfigurable atomic memory for dynamic networks. In *Proc. of the Intl. Conference on Dependable Systems and Networks*, pages 259–269, June 2003.

[12] J. Hill, R. Szewcyk, A. Woo, D. Culler, S. Hollar, and K. Pister. System architecture directions for networked sensors. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

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

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

[15] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, pages 233–242. ACM Press, 2002.

[16] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[17] Nancy Lynch and Alex Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. of the 16th Intl. Symp. on Distributed Computing*, pages 173–190, 2002.

[18] Nancy Lynch and Alex Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. Technical Report LCS-TR-856, M.I.T., 2002.

[19] Samuel R. Madden, Robert Szewczyk, Michael J. Franklin, and David Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Workshop on Mobile Computing and Systems Applications*, 2002.

[20] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, and David Culler. Wireless sensor networks for habitat monitoring. In *2002 ACM International Workshop on Wireless Sensor Networks and Applications*, September 2002.

[21] Roberto De Prisco, Alan Fekete, Nancy A. Lynch, and Alexander A. Shvartsman. A dynamic primary configuration group communication service. In *Proc. of the 13th Intntl. Symposium on Distributed Computing*, pages 64–78, September 1999.

[22] Tim Shepard. Personal Communication.

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

A Execution Data

Excerpt from an experimental trace, showing 43 read and write operations.

Time	Particle UID	Event	Value	Time	Particle UID	Event	Value
343	0.7176	start write	0.2927	469	0.3265	start write	0.2420
344	0.8858	start read		469	0.7742	start read	
353	0.7176	end write		470	0.2794	start write	0.0307
354	0.8858	end read	0.7969	474	0.3998	start write	0.4478
360	0.6779	start read		480	0.7742	end read	0.1217
362	0.4234	start write	0.5990	481	0.3265	end write	
362	0.9121	start read		481	0.2794	end write	
363	0.3304	start read		483	0.9033	start read	
368	0.6779	end read	0.2927	487	0.3998	end write	
370	0.1527	start read		494	0.9033	end read	0.4478
370	0.4234	end write		506	0.8641	start read	
372	0.9121	end read	0.2927	508	0.3271	start read	
374	0.3304	end read	0.5990	511	0.1527	start read	
375	0.9033	start write	0.5304	512	0.7176	start write	0.3067
382	0.7742	start read		517	0.8641	end read	0.4478
382	0.5729	start read		517	0.1244	start write	0.7057
384	0.1527	end read	0.5990	520	0.1527	end read	0.4478
384	0.9033	end write		522	0.3271	end read	0.4478
388	0.5729	end read	0.5304	523	0.0653	start write	0.1060
389	0.7742	end read	0.5304	523	0.7176	end write	
390	0.1530	start write	0.5531	529	0.1276	start write	0.0150
400	0.3463	start read		530	0.1410	start read	
403	0.1530	end write		530	0.3271	start write	0.7464
411	0.3463	end read	0.5531	531	0.1244	end write	
412	0.8858	start read		531	0.0982	start read	
418	0.6958	start write	0.9094	532	0.8858	start read	
418	0.5729	start read		534	0.0653	end write	
420	0.8858	end read	0.5531	537	0.8241	start read	
422	0.5541	start read		540	0.1276	end write	
427	0.5729	end read	0.5531	541	0.6496	start write	0.3561
429	0.6958	end write		542	0.5054	start write	0.7986
429	0.6827	start write	0.9085	543	0.1276	start read	
433	0.6692	start read		544	0.1410	end read	0.1060
434	0.5541	end read	0.9094	544	0.8858	end read	0.01502
435	0.5541	start write	0.0646	545	0.3271	end write	
436	0.3265	start read		545	0.0629	start read	
438	0.6827	end write		547	0.0982	end read	0.7464
441	0.1244	start write	0.7466	549	0.3998	start write	0.2434
445	0.3265	end read	0.9085	551	0.8241	end read	0.7464
446	0.6692	end read	0.9085	555	0.1276	end read	0.7464
446	0.5541	end write		556	0.5541	start read	
447	0.8858	start write	0.9352	558	0.6265	start write	0.2072
451	0.9496	start write	0.1217	558	0.6496	end write	
455	0.1244	end write		560	0.0982	start write	0.3324
459	0.8858	end write		561	0.0629	end read	0.3561
462	0.9496	end write					