massachusetts institute of technology — artificial intelligence laboratory

Persistent Nodes for Reliable Memory in Geographically Local Networks

Jacob Beal

AI Memo 2003-011 April 2003

Abstract

A Persistent Node is a redundant distributed mechanism for storing a key/value pair reliably in a geographically local network. In this paper, I develop a method of establishing Persistent Nodes in an amorphous matrix. I address issues of construction, usage, atomicity guarantees and reliability in the face of stopping failures. Applications include routing, congestion control, and data storage in gigascale networks.

1 Introduction

I want to put pieces of data into a network and store them reliably by means of enormous redundancy. That redundancy should be achieved by distributing a huge number of copies of the data over a geographically local portion of the network, so that maintaining consistency can be fast and cheap. Conversely, each node in the network should be able to help store many pieces of data. Persistent Nodes are a mechanism for doing this, and I explore their implementation on an amorphous matrix. Key results include atomicity for failure-free executions and for executions with limited stopping failures, as well as experiments demonstrating atomicity in simulation.

2 Amorphous Computing Model

One useful model for studying this domain is two-dimensional amorphous computing.[2] An *amorphous network* is generated by distributing n particles uniformly randomly over a unit square (or other 2D region) and placing edges between all pairs of particles less than distance d apart. The resulting graph has spatially local connectivity but no long-distance links. The *amorphous matrix* for a given network refers to the spatial embedding of the network graph.

Each particle has a clock which drifts at a rate $|r| \leq \epsilon$ per tick, yielding timing uncertainty $L = \frac{1+\epsilon}{1-\epsilon}$. On each particle, we run an identical partially synchronous algorithm, subject to the following restrictions:

1. Particles receive no geometric information or coordinates
2. Particles receive no time synchronization information
3. Each particle's clock is initialized to a random time

Any of a wide variety of communication models may be applied. For purposes of this paper, I assume send/recieve communication over links with reliable message delivery in maximum time t_m.

3 Technical Sketch of Persistent Nodes

A Persistent Node is, fundamentally, just a key/value pair that lives in a location: imagine that we spray-paint a red circle on the amorphous matrix and declare that everything painted red will remember a piece of data. More precisely, a node N is a circular area of radius r hops on the amorphous matrix in which the particles store value X and define operations $READ(N, X)$ and $WRITE(N, X)$. Besides being a READ/WRITE object, however, a node has several other useful properties.

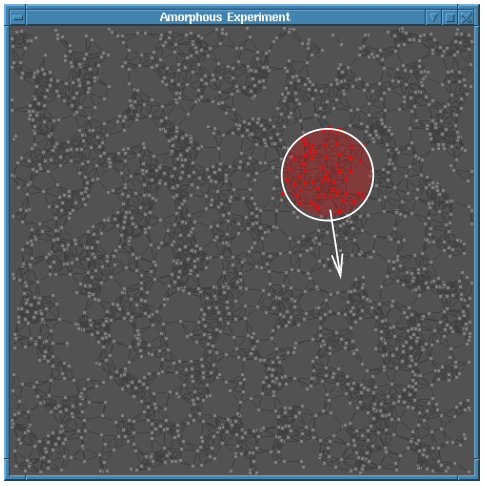


Figure 1: A Persistent Node (red) on an amorphous matrix (grey). The particles composing the node occupy a spherical region in space. The arrow shows direction of motion for the node, toward a denser region of particles.

First and foremost, node state is local in extent. This is because we want the matrix's capacity scale linearly with its area, so nodes need to consume constant resources. Thus, information about a node N is confined to $3r$ hops distance from its center.

In a geographically local network, failures are likely to be correlated in a region (loss of power to an entire neighborhood, for example), wiping out regions of the amorphous matrix. Thus, a node is designed to tolerate stopping failures of regions as well as points. Every particle of a node contains a copy of the node's value, and can serve as a seed to reconstruct the node. If any particle in the node does not fail, and is connected to the rest of the network via non-failing particles, then the surviving particles will recruit other particles to join the node, rapidly restoring its lost redundancy.

Nodes can also move from place to place in the matrix. Nodes move by shifting their centers incrementally, with the direction of motion determined by a potential function Φ. Φ can be varied to obtain different effects: one application of this mobility is to scoot away from edges in the matrix — such as the border of a failure region. Another example is congestion control, where nodes in a heavily utilized area of the matrix diffuse toward less burdened areas.

Finally, a node is locally indistinguishable from a READ/WRITE atomic object,[1] and can be transformed into a true READ/WRITE atomic object.[11] Many applications, such as routing, do not need nodes to be atomic, and atomicity is costly, so by default persistent nodes fulfill only the weaker condition of local atomicity. Atomic behavior is necessary for other applications, such as data storage, so it is equally important that there be a reasonable method for transforming a node into an atomic object.

In the subsequent sections, I will explain the PersistentNode algorithm in detail. In Section 4, I describe how a node is created and maintained, then how it moves. In Section 5, I describe how nodes execute READ and WRITE operations. In Section 5.1 I prove local atomicity for the static failure-free case and demonstrate how it can be converted to atomicity. In Section 5.2, I extend the proofs to the dynamic case — moving nodes, and finally in Section 6 I discuss the algorithm's

[1] Speaking informally, this extremely weak property looks to capture the benefits of both the *eventually serializable* property in [7] and the *weak sequential locking* in [12].

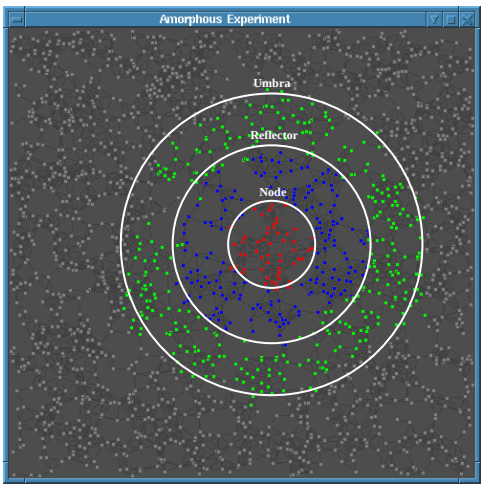


Figure 2: Anatomy of a persistent node. The innermost circle is the node proper (red), where data can be read or written. Every particle within middle circle is in the reflector, which holds data and calculates which direction the node will move. The outermost circle is the umbra, which knows the name of the node, but nothing else.

behavior under stopping failures and prove atomicity for limited region failures.

4 Building and Moving Nodes

To create a node, we invoke $CREATE(N, r, \Phi)$ on a particle. This particle becomes the center of a node named N with radius r and potential function Φ. It now begins the node life-cycle: search outward to define the node, collect heuristic values inward to choose a successor, and hand off the duty of being center to a successor.

The center of a node continually runs a depth-limited breadth first search to define the node (this is essentially the gradients used by Nagpal and Coore [3][15][16]). The BFS labels all particles less than $3r$ hops from the center with their hopcount from the center (particles with higher hopcounts unlabel themselves). A particle p with hopcount $h(p)$ equal to c has a set $parents(p)$ of neighbors with hopcount $c-1$ and $children(p)$ of neighbors with hopcount $c+1$. Particles with $h(p) < r$ are in the node proper. Particles with $h(p) < 2r$ are in the *reflector* — the region which controls movement direction, and particles with $h(p) < 3r$ are in the *umbra*. (See Figure 2)

Every particle in the reflector (which includes all particles in the node) participates in calculating the direction of a move. At every time step, each reflector particle calculates its current heuristic value $v(p)$ and transmits it to its parents. The heuristic is a sum of partial paths through p, multiplied by Φ (which is generally ≤ 1):

$$v(p) = \Phi(p) \cdot (1 + \sum_{i \in children(p)} v(i))$$

This heuristic has the effect of giving higher values to denser and more well-connected regions. Particles at the edges of the reflector (where the graph approximates its underlying geometry well, as shown in [9]) are counted the greatest number of times, but particles nearer the center go through less Φ reductions. With the function $\Phi_0(p) = 1$, particles in the direction of a nearby edge of the matrix will receive smaller values than interior particles. (Figure 4)

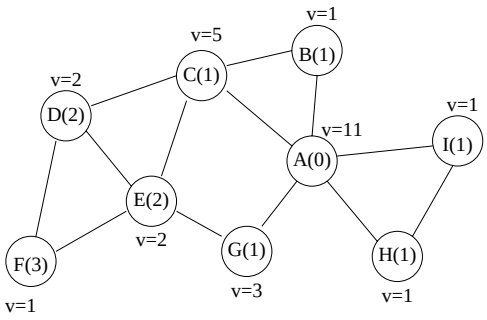
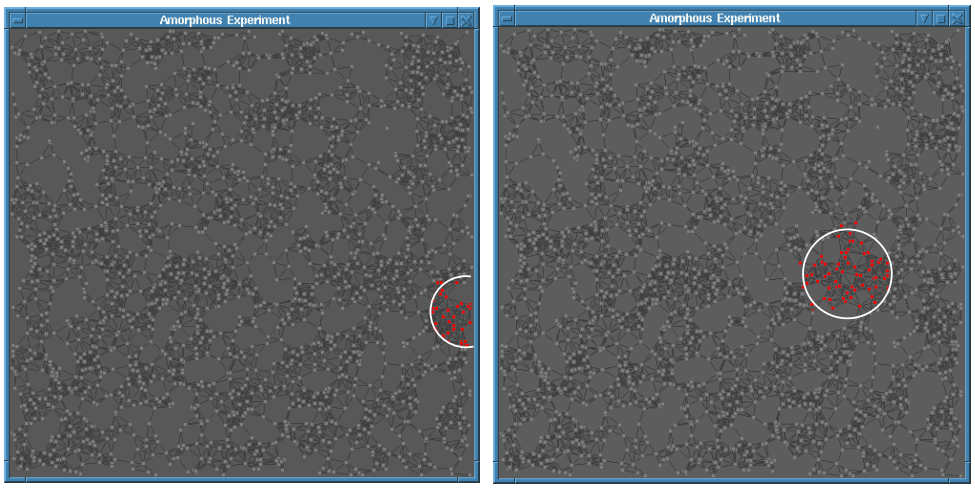


Figure 3: The heuristic value of a particle is the 1 plus the sum of the values of its children, multiplied by the potential function Φ. In this example, $\Phi(p) = 1$, and every particle is labelled Name(Distance from Center). Note that particle F is counted three times. The center (A) will select C as its successor, since C has the highest heuristic value of its neighbors.



(a) Start (n=0) (b) Finish (n=10)

Figure 4: The heuristic value function causes Persistent Nodes to move away from edges (or damaged regions) toward regions with high particle density. In this example, a radius 4 node started with its center at the edge moves rapidly towards the interior of the amorphous matrix, moving more than a full diameter in 10 steps.

If a particle x is in the node, has no parents, and its heuristic value has not changed for T_s rounds, then x chooses a good successor from its neighbors based on their heuristic values — in failure-free executions, x will always be the current center, as it is the only particle naturally without a parent, and convergence will take at most $4Lr$ rounds (time for a message to propagate to the edge of the reflector and back). To combat local minima in Φ, x randomly chooses the neighbor with nth highest value with probability $(1-p_k)p_k^{n-1}$.

The newly designated successor of x now initiates a new round of searches as x becomes quiescent, replacing the results of the previous search[2] and beginning the process again, only with the node's center shifted slightly.

5 Read/Write Operations on Nodes

To be able to tolerate stopping failures, a persistent node storing value X keeps a copy of X at every particle in the reflector (particles which leave the reflector dump their copies of X). Only particles in the node will accept $READ(N, X)$ and $WRITE(N, X)$ operations,[3] though all particles in the reflector store copies of the data — this is to allow the consistency properties to hold even in a moving node.

The copies are kept consistent with a logical sequence of READ and WRITE operations by means of a version tag of the data. The version tag is a pair $V = (v, i)$ of version number and particle ID, where the particle ID is used for tie-breaking. Every round, each particle sends its data and tag to its neighbors; the neighbors compare the incoming state to their own and set their new state to the copy with the greater tag.

Initially, every particle contains a null value with tag $(0, 0)$. A $WRITE(X)$ on particle p sets its data to X and its tag to $(v + 1, p)$, then sends an *ack* immediately. This data propagates one hop per round throughout the rest of the particles in the reflector, until either every particle shares state $(X, v+1, p)$ or it is replaced by a bigger version tag. Thus, for a node of radius r, the WRITE will propagate throughout the node in $< 2r$ rounds and throughout the reflector in $< 3r$. The duplication makes READ operations simple: they just return the local copy of X immediately. (If we wish, a READ operation can return the version tag as well, for use in building end-to-end atomic systems.)

I now want to prove the logical atomicity property and show that it can be used to produce atomicity. For now, I consider only failure-free executions.

5.1 Static Read/Write

We begin with the static case, in which the center of the node is not moving. This allows us to assume a static set of participants in the algorithm and that the subgraph of those participants is connected. First, we extend the version tags to be a logical time assignment, using the LamportTime transformation.[10] Note that since READ and WRITE operations are instantaneous, we analyze them as single events rather than separating them into initiation and response.

Theorem 1: *Let R be the node algorithm, and L(R) be the* LamportTime *transformation of R.*

[2]Search instances are tagged with a monotonically increasing version number that uses the initiator's heuristic value and process ID as tie-breakers; this is used for resolution between competing successors as well as handing off control to single successor.

[3]Henceforth, I will leave off the parameters for READ and WRITE operations, since operations on two nodes are orthogonal and we thus need only discuss operations on a single node N.

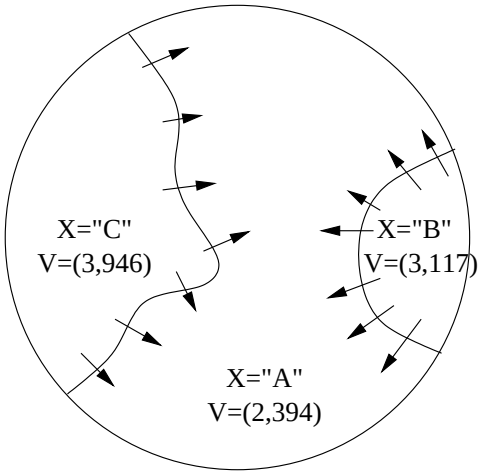


Figure 5: WRITE operations add one to the current version and propagate. If more than one WRITE operation is initiated with the same version, the particle ID where the operation was invoked is used to disambiguate. In this example, the old value A is being overwritten by two concurrent writes, B and C, with the same version number. When C and B encounter each other, C will overwrite B because its particle ID is bigger.

The ith event E_i occurs on particle p_i. If t_i is the logical time assignment assigned by L(R) for event E_i and V_i is the version tag of the data at p_i when E_i occurs, then (V_i, t_i) is a logical time assignment under lexicographic order.

Proof: A logical time assignment has four properties:

1. No two events have the same logical time.

This property is inherited directly from LAMPORTTIME, since every t_i is unique.

2. The logical times of a process's events are strictly increasing.

The version V_i of a process is non-decreasing. If the version is the same for two successive events, then comparison falls to its t_i, which LAMPORTTIME guarantees are strictly increasing.

3. The logical time of sends are smaller than receives.

There are three cases when E_i and E_j are a send/receive pair (in that order):

1. If V_i is less than V_j, then the order is correct.
2. If V_i is equal to V_j, then LAMPORTTIME guarantees that t_i is less than t_j
3. If V_i is greater than V_j, then R sets V_j to V_i, reducing it to case 2.

4. For any logical time t, only a finite number of events smaller than t are assigned.

For a logical time (V_i, t_i), LAMPORTTIME guarantees that there are only a finite number of elements t_j less than t_i. We need only worry, then, about elements (V_j, t_j) where t_i is less than t_j and V_i is greater than V_j. In any fair execution on a connected graph, however, only a finite number of events can occur before a sequence of sends and receives which chains to every particle with version V_j, thereby increasing its version to V_i.

Thus, (V_i, t_i) is a logical time assignment. □

We now show that, under this logical time assignment, reads follow writes, then use that to demonstrate local atomicity.

Lemma 2: *Consider a WRITE event E_i with logical time (V_i, t_i) and READ event E_j with logical time (V_j, t_j). If $V_j = V_i$, then $t_i < t_j$.*

Proof:

Data can't be read from a particle before it arrives at that particle: thus either a WRITE or a receive event of data with version V_i occurs before the read. In the case of a WRITE event, since versions are not duplicated it must be E_i. Then E_i and E_j occur on the same particle, and therefor $t_j > t_i$ by the second property of logical time assignment. Similarly for data arriving via receive event: the third property of logical time assignment means there is a chain of send and receive events from the particle where E_i occurs to the particle where E_j occurs, all with strictly increasing logical times $t_i < t_{s_0} < t_{r_0} < ... < t_{r_k} < t_j$. Thus in both cases $t_i < t_j$. □

Theorem 3: *Every execution α of R is locally indistinguishable from an execution α' which satisfies the atomicity property.*

Proof:

Since (V_i, t_i) is a logical time assignment, we can transform α into a locally indistinguishable execution α' where events occur in the order of their logical times. Since the logical time of a write is before that of any read sharing the same version, and all reads of a given version occur between the write of that version and the write for the next (Lemma 2), then we can simply set the serialization points at the invocation of each operation and produce a sequence with the atomicity property. □

This local atomicity property can be transformed to make the node a READ/WRITE atomic object by delaying the response events for READ and WRITE operations. This can be done in the asynchronous case if every particle keeps a *knowntime* record for all other particles in the node, and delays responses until its record indicates that no particle has a lower V_i than it did at invocation. This is very costly, however, and stronger than necessary.

If we weaken the algorithm to depend on partial synchrony, then it is much less expensive. Instead of maintaining group membership lists, the transform PSynchDelay(A,R) delays responses to READ and WRITE operations until $3Lr$ rounds have passed, where r is the radius of the node, L is the timing uncertainty and a round is long enough for all messages to be delivered.

Theorem 4: PSynchDelay(R,R) *is atomic under a partially synchronous execution with time uncertainty L.*

Proof:

We will use Lemma 13.16 from Lynch[13] to prove this. For any execution β of PSynchDelay(R,R), let the ordering $\prec$ of the operation sequence Π be the invocation events' logical time assignment (V_i, t_i), as defined in Theorem 1, with the modification that precedence relations between pairs of READ operations with the same V_i are omitted. We now show that it satisfies the required properties:

1. For any operation x in Π, there are only finitely many operations which preceded it.

This is directly inherited from property 4 of logical time assignment.

2. If the response event for x precedes the invocation event for y in β, then it cannot be the case that $y \prec x$.

This is the case which the unmodified algorithm fails, requiring the delay imposed by PSynchDelay.

In all cases, the critical property is the time delay $D = 3Lr$, which is 3/2 the (skewed) maximum time delay to propagate a message between any two particles in the node, and equals to the

maximum delay to propagate a message between a particle in the node and a particle in the reflector. Because particles update their data to match neighbors on every receive, then it must be the case that if a particle has version V_i at time t, then after it counts an additional D cycles, every other particle in the reflector must have a version of at least V_i.

Thus, since x's response occurs before y's invocation, the delay guarantees that it must be the case that $V_x \leq V_y$. If $V_x < V_y$, then $x \prec y$. If V_x and V_y are equal, then if both operations are READs, no precedence relation holds between them. If both operations are WRITEs, then V_x and V_y cannot be equal. If one is a READ and the other is a WRITE, then by Lemma 2 either V_x and V_y are unequal or $x \prec y$.

3. If x is a WRITE in Π, and y is another operation in Π, then either $x \prec y$ or $y \prec x$.

This is implied by the inequality of logical time assignments.

4. The value returned by a READ is the value written by the preceding WRITE.

By Lemma 2, we know that for every READ with invocation time (V_i, t_i), any WRITE with invocation time (V_i, t_j) precedes it in $\prec$. Since V_i is only increased by WRITE operations, and a READ (V_i, t_i) returns the value associated with V_i, this implies that each READ returns the value written by the previous WRITE.

Thus β satisfies the atomicity property. □

5.2 Dynamic Read/Write

I now extend these results into the dynamic case, where the center of the node is moving. For this case, the participation of the reflector becomes important: intuitively, this involves showing that the node cannot move fast enough to overrun the area of the reflector in which the properties proved above hold.

First, we need to formalize how the READ/WRITE portion of the node algorithm relates to the mobility portion of the node algorithm. In both cases, each particles sends its neighbors a state message every round. We will assume that these two state messages are sent as a single combined message, thereby avoiding a number of messy race cases.

Next, we need to determine how fast the center of a node is capable of moving with respect to its reflector: I find a general formula for the distortion of a node, and three correlates which apply the general formula to produce convenient quantities.

Theorem 5: *Let $h_0(p)$ be the minimum distance from the center of a moving node to some particle p in its reflector. The difference of $h_0(p)$ from the hopcount $h(p)$ stored at p is at most $\frac{(h(p)+1)L}{T_s}$, provided that $T_s > L$*

We note that the center of a node moves at most once every T_s rounds (in practice, the rate is actually more like $2r + T_s$, but we are considering a worst case). To obtain the worst skew, we run the center particle at maximum speed and the rest of the node at minimum speed, producing a timing uncertainty of L between the center particle and every other particle. The hopcount $h(p)$ begins at $h_0(p)$, and the center is moving directly towards or away from it at its maximum rate of one hop per change of center.

If the center has moved away from p, then propagating the BFS from the center takes $h(p) + 1$ rounds, for a maximum time delay of $(h(p) + 1)L$ center rounds for a change at the center to propagate to the edge. During this time, the center can move once every T_s rounds, for a maximum displacement of $\frac{(h(p)+1)L}{T_s}$. If $T_s > L$, then subsequent changes do not stretch this bound any further, because the propagation times overlap. □

Note that this property is only well-defined for nodes that have $h(p) < 3r$, which translates to being guaranteed well-defined only for $h_0(p) < 3r - \frac{3rL}{T_s}$.

Corollary 6: *If $T_s \geq 4L$, then the maximum displacement of the center with respect to the node will be $\frac{1}{4}r$, the maximum displacement with respect to the reflector will be $\frac{1}{2}r$, and the maximum displacement with respect to the umbra will be $\frac{3}{4}r$.*

Obtained by plugging in the reflector bound $2r - 1$ for $h(p)$. The property is well defined over all nodes with $h_0(p) < 2r - 1$ because $\frac{3rL}{T_s} = 3/4$. A similar calculation produces the node and umbra bounds.

Corollary 7: *Assuming $T_s \geq 4L$, the maximum diameter of a node is $\frac{9}{4}r$ and the maximum diameter of a reflector is $\frac{9}{2}r$*

Obtained by plugging in the node radius bound $r - 1$ for $h(p)$. Since this maximal movement is in one direction, we only need to add $\frac{1}{4}r$ once, not twice. A similar calculation produces the reflector bound.

Corollary 8: *Assuming $T_s \geq 4L$, every particle with $h_0(p) < \frac{3}{4}r - 1$ is a member of the node, every particle with $h_0(p) < \frac{3}{2}r - 1$ is a member of the reflector, and every particle with $h_0(p) < \frac{9}{4}r - 1$ is a member of the umbra.*

Applying a little algebra to the displacement formula, we obtain $h(p) \leq \frac{4}{3}h_0(p) + \frac{1}{4}$. Setting $h(p)$ to the minimum distance no longer in the node, r, we determine that only particles with $h_0 \geq \frac{3}{4}r - \frac{3}{16}$ may be outside of the node. A similar calculation produces the reflector and umbra bounds.

We now augment our analysis of the static execution by adding two new serial operations, $REFLECTOR_i$ and $NODE_i$, which model movement of the node in the PersistentNode algorithm. Particle i's participation in the reflector is denoted by the $REFLECTOR_i$ operation, and its participation in the node is denoted by the $NODE_i$ operation. Every $NODE_i$ operation is contained within a $REFLECTOR_i$ operation, and only one $NODE_i$ and $REFLECTOR_i$ operation can be in progress concurrently for a given particle.

All events for particle i occur between the invocation and response events of a $REFLECTOR_i$ operation, and all invocations of READ and WRITE operations occur between the invocation and response events of a $NODE_i$ operation.

Finally, since particles flush state when not in the reflector, we will analyze each $REFLECTOR_i$ operation as a different process on the particle. This is necessary to deal with a case when a particle leaves and rejoins a node during a write, leaving with the new value and begin reinitialized with the old, and need not introduce any inconsistency. We will call executions which obey these constraints *dynamic executions*, and extend our previous proofs to cover them.

Theorem 9: *Let α be a dynamic execution of R. The labelling (V_i, t_i) defined in Theorem 1 is a logical time assignment.*

Proof: We follow the same proof sketch as for Theorem 1, showing the four properties of a logical time assignment:

1. No two events have the same logical time.

Proof as for Theorem 1.

2.: The logical times of a process's events are strictly increasing.

The proof in Theorem 1 requires some small augmentation. The invocation and response events of a $NODE_i$ operation have no effect on V_i, and therefor operate as for any other event. For a $REFLECTOR_i$ operation, the invocation and response events are labelled normally by LamportTime, and have respectively the minimum and maximum versions V_i, which is consistent

with a strictly increasing sequence.

3. The logical time of sends are smaller than receives.

Not all sends are paired with a receive: a message which arrives at a particle between $REFLECTOR_i$ operations is dropped, which is consistent. Otherwise, the proof proceeds as for Theorem 1.

4. For any logical time t, only a finite number of events smaller than t are assigned.

The connectedness assumption no longer holds as per Theorem 1. By Corollary 8, however, we know that at all times there is a connected region of reflector around the center with at least radius $\frac{3}{2}r - 1$. The maximum radius of the node is $\frac{5}{4}r$, which means that no particle in the node can be disconnected from the center.[4] Within the connected region of the reflector, the chaining from Theorem 1 applies, limiting the number of events occurring before (V_i, t_i).

Disconnected regions of the reflector cannot include any particles in the node, so no READ or WRITE events can occur, leaving the only possibility of infinite events as message events. However, the message events also carry updates to $h(p)$, and thus, under partial synchrony, at most Lr rounds of send and receives can occur in a disconnected particle before a chain of sends and receives from the connected region of the reflector either updates its data or concludes its current $REFLECTOR_i$ operation. □

Lemma 10: *In a dynamic execution of R, consider a WRITE event E_i with logical time (V_i, t_i) and READ event E_j with logical time (V_j, t_j). If $V_j = V_i$, then $t_i < t_j$.*

Proof: The proof used for Lemma 2 holds for dynamic executions as well, since $V_j = V_i$ implies the existence of a chain of send and receive events between the two particles.

Theorem 11: *Every dynamic execution α of R is locally indistinguishable from an execution α' which satisfies the atomicity property.*

Proof: The proof used for Theorem 3 holds, given the logical time assignment proved in Theorem 9, and the read/write ordering in Lemma 10.

Finally, we extend the atomicity proof for PSynchDelay(R,R) to the dynamic case:

Theorem 12: PSynchDelay(R,R) *is atomic under a partially synchronous dynamic execution with time uncertainty L and $T_s \geq 4L^2$.*

Proof: To extend the proof in Theorem 4 to the dynamic case, we use Lemma 13.16 from Lynch as before, and additionally show the well-formedness condition that READ and WRITE operations respond during the $REFLECTOR_i$ operation in which they were initiated.

Starting with the well-formedness condition: READ and WRITE operations can only be invoked during a $NODE_i$ operation. To show well-formedness, then, we must show that the time between the termination of a $NODE_i$ operation and its enclosing $REFLECTOR_i$ operation is at least $3L^2r$. In order for a particle to leave the reflector, $h(p)$ must be at least $2r$. In order for a particle to be a part of the node, $h(p)$ can be at most $r - 1$, meaning that $h_0(p)$ is at most $\frac{5}{4}r - 1$ by Corollary 6, and that no more than $\frac{1}{4}r$ increases in $h(p)$ can be pending as the time of the invocation event.

Thus the center must move away from the particle an additional $\frac{3}{4}r + 1$ hops in order to drop $h(p)$ enough to remove the particle from the reflector. Since the center moves with maximum velocity $\frac{1}{T_s} = \frac{1}{4L^2}$, the time between the termination of a $NODE_i$ operation and its enclosing $REFLECTOR_i$ operation is at least $3L^2r$. Thus, we have well-formedness.

[4] To allow disconnected particles in the node, the connected region of the reflector must be at least one hop smaller than the maximum radius of the node.

Next, we prove the properties required by Lemma 13.16, for which we use the looser bound of $T_s \geq 4L$:

1. For any operation x in Π, there are only finitely many operations which preceded it.

Same as for Theorem 4.

2. If the response event for x precedes the invocation event for y in β, then it cannot be the case that $y \prec x$.

If $3Lr$ is sufficient time to propagate a message across the node, then the same proof as in Theorem 4 will hold. Assuming $T_s \geq 4L$, the maximum diameter of a node is $\frac{9}{4}r$ by Corollary 7, and the maximum velocity of its center is $1/T_s$, while the minimum velocity of a message is $1/L$. If the direction of message travel is the same as the direction of travel of the center, then the velocities will subtract, producing the minimum effective message velocity, $\frac{1}{L} - \frac{1}{T_s} = \frac{3}{4L}$. Plugging in the distance and velocity equations, we find that in fact a message will travel across the node in time $3Lr$, thereby ensuring that every particle in the reflector has version at least V_i before the response event of x. The rest follows as in Theorem 4.

3. If x is a WRITE in Π, and y is another operation in Π, then either $x \prec y$ or $y \prec x$.

Same as for Theorem 4.

4. The value returned by a READ is the value written by the preceding WRITE.

Same as for Theorem 4.

Thus PSynchDelay is atomic for the dynamic case as well. □

6 Stopping Failures

Finally, we need to consider the effect of stopping failures on the system. Because of the spatial embedding of the amorphous matrix, we need to consider region failure cases (e.g. a neighborhood losing power) as well as standard f failure cases (which can be viewed as special cases of single-particle regions failing)

Intuitively, the PersistentNode algorithm will deal well with most failures. If any particle in the node survives a failure, then eventually it will choose a successor to become the new center, which starts the node construction cycle again. Similarly, any failure which does not disconnect the reflector will allow WRITE operations to propagate their values throughout the region, albeit possibly slower than previously.

Problems arise when a failure disconnects two portions of the node. In these cases, we can end up with two separated copies of the node maintaining different values. If this is due to a global partition of the network, then consistent behavior is impossible since there is no way for information to pass between the two copies of the node. Thus, we do not deal with this case.

If the partition is local in scope, however, then whether the node is duplicated depends on whether its two halves are outside of the range of each others' construction process. If the two copies of the node are close enough, then their construction searches will touch and one will have a larger tag, causing all of the particles in the other copy to be labelled with hopcounts greater than $3r$ and thereby deleted. If, on the other hand, their are $6r$ distant, then the node will be duplicated, which is not good.

For cases where the two copies are too far apart to resolve the conflict on their own, it is possible to build a global infrastructure which detects and resolves duplication conflicts and guarantee atomic behavior over a wider range of failures; I will not, however, describe such a system in this paper other than to note that it can be constructed.

For local failures of small radius, however, PSynchDelay(R,r) is atomic if we assume a slower maximum velocity for the movement of the center:

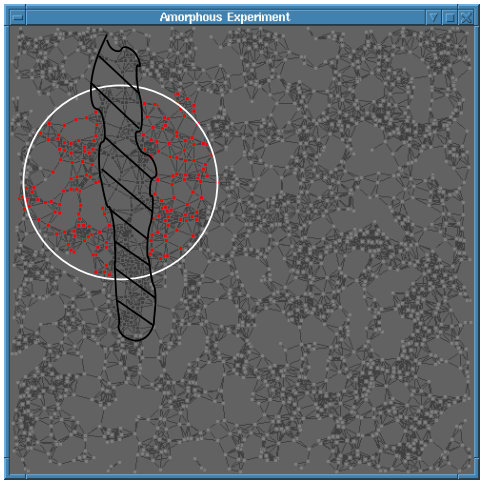


Figure 6: Stopping failures which partition the node, but not the graph, are difficult to recover from. In this example, the dark nodes have failed, partitioning the large red node into a left and right half. Atomicity is not preserved by *PSynchDelay* in this case, although interim atomicity is, because the partition is small enough for the umbras of the two halves to touch.

Theorem 13: *Assume* $T_s \geq 8L^2$. PSynchDelay(R,R) *is atomic under a partially synchronous dynamic execution with time uncertainty* L *and stopping failures which increase the diameter of the node subgraph by no more than* $\frac{1}{2}r$ *and which displace the center by no more than* $\frac{1}{2}r$

Proof: The proof for this theorem closely follows the proof for the dynamic case in Theorem 12.

First, we demonstrate well-formedness: the larger T_s reduces the distortion of the node so that at the invocation of a READ or WRITE on particle p, its distance from the center is $h_0(p) \leq \frac{9}{8}r - 1$ (Theorem 5), and therefor no more than $\frac{1}{8}r$ increases in $h(p)$ can be pending at the time of the invocation event, and the center must move away $\frac{7}{8}r + 1$ hops to drop $h(p)$ enough to remove the particle from the reflector. The center moves at maximum velocity $\frac{1}{T_s} = \frac{1}{8L^2}$ following a possible initial leap outwards of displacement x if a region including the old center failed. Solving for x, we obtain $\frac{1}{2}r$ as the maximum displacement for which well-formedness is guaranteed.

Next, we wish to use Lemma 13.16 to prove atomicity (relaxing the bound to $T_s \geq 8L$), but have executions which may include incomplete operations. To cope with this for an execution β, we produce a transformed execution β' in which there are no incomplete WRITE operations. If x is an incomplete WRITE in β, then if no *receive* event ever transmits the value of the write to a particle in the node which does not fail before $3Lr$ has elapsed, then invocation event for x is removed from β'. Otherwise, a response event for x is inserted in β' immediately before the failure which terminated x.

In effect, this transformation assigns a serialization point to every incomplete WRITE event, and removes the event from the operation sequence Π' if the serialization point is after the particle fails. If β' has the atomicity property, then β has the atomicity property as well. We now show that Lemma 13.16 holds for β':

1. For any operation x *in* Π*, there are only finitely many operations which preceded it.*

Same as for Theorem 12 and Theorem 4.

2. If the response event for x *precedes the invocation event for* y *in* β*, then it cannot be the case that* $y \prec x$*.*

This follows the same sketch as for Theorem 12.

Since $T_s \geq 8L$, the maximum velocity of the center is $1/8L$, while the minimum velocity of a message is $1/L$. Minimum effective message velocity is $\frac{7}{8L}$, and if the message travels across the node in less than $3Lr$, the well-formedness condition and the diameter limit guarantee that it will able to reach every portion eventually. Multiplying velocity and time, we find that in $3Lr$ a message will be able to reach every particle in a node with an effective diameter of at most $\frac{21}{8}r$. A failure-free node has maximum diameter $\frac{17}{8}r$ by Theorem 5, so if the diameter increase caused by failures is bounded by $\frac{1}{2}r$, then a message will travel across the whole node within $3Lr$. The rest follows as in Theorem 12 and Theorem 4.

3. If x is a WRITE in Π, and y is another operation in Π, then either $x \prec y$ or $y \prec x$.

Same as for Theorem 12 and Theorem 4.

4. The value returned by a READ is the value written by the preceding WRITE.

If we can prove that a WRITE in β but not β' is never returned by any READ, then it reduces to the case of Theorem 12 (incomplete READ operations may be ignored since they never return).

A WRITE invocation event in β does not appear in β' if no particle ever receives the value which does not fail before $3Lr$ elapses following the invocation event. Any particle which fails before $3Lr$ cannot have completed a READ event started after the WRITE operation was invoked, and by assumption no particle which has time enough to complete a READ event can have ever received the value. Thus, no READ event can ever return the value of a WRITE not in β'.

The rest follows the same proof as for Theorem 12 and Theorem 4.

Thus PSynchDelay is atomic for failures which increase diameter by less than $\frac{1}{2}r$ and displace the center by less than $\frac{1}{2}r$. □

Corollary 14: *Assuming $T_s \geq 8L$,* PSynchDelay(R,r) *is atomic under a partially synchronous dynamic execution with time uncertainty L and stopping failures confined to an n-sphere with radius $\frac{r}{2\pi}$*

The circumference of such an n-sphere is r, meaning that it must add less than $\frac{1}{2}r$ to the diameter of the node. If it causes maximum displacement of the center, that can be no farther than its diameter, $\frac{1}{\pi}r$, which is also small enough to fit the requirement of Theorem 13. □

Finally, the weaker property of interim atomicity applies for all failures that do not result in node duplication. Interim atomicity, as defined in [8], states that an algorithm is atomic except during unstable periods (defined portions of the execution which contain failures) and for a finite period of time following an unstable period.

Theorem 15: PSynchDelay *is interim atomic for failures which do not result in node duplication.*

If any particle in the node survives, then the node exists in at least one place. If the node is not duplicated, then no other copy exists within range of the node's BFS. Since the construction process is continuous, then within $3Lr$ the node will be reconstructed with precisely one center, reducing it to the dynamic case, which is atomic. □

Theorem 16: PSynchDelay *is interim atomic for any execution in which, at the end of the unstable period, the distance between any two particles in the node is less than $\frac{5}{2}r$.*

Proof: Following an unstable period at real time t_s, some k particles will anoint a successor. We label these particles in order as $p_0, p_1, ..., p_k$ and they occur at times $t_s < t_0 < t_1 < ... < t_k$. The times are restricted by the convergence process, so $T_s < t_0 - t_s < t_k - t_s < 4Lr + T_s$. Once a particle has anointed a successor to become the node center, that successor can move at a maximum velocity

of $1/T_s$.

From Corollary 8, assuming $T_s \geq 4L$ we have that every particle with $h_0(p) < \frac{9}{4}r - 1$ is a member of the umbra, and round this down to $2r$ for simplicity (valid for $r > 3$). This guarantees that one of the two copies of the node will be eliminated if their centers are less than $4r$ apart. The time for a BFS to reach the $2r$ threshold is at most $2Lr$, so the maximum skew between when p_0 anoints a successor and p_k's first BFS reaches the threshold is $6Lr$. Assuming $T_s \geq 4L$, then the maximum distance that any p_i can have traveled is $\frac{3}{2}r$ when p_j's BFS reaches the threshold for elimination. Thus, elimination of all duplicates will eventually occur whenever no pair of p_i and p_j are more than $\frac{5}{2}r$ hops apart, providing interim atomicity by Theorem 14. □

6.1 Adding Particles

In a large enough network, we should expect nodes to be added as well as removed. These additions model both the addition of new particles and the rebooting or replacing of failed particles. Both cases are handled simply and identically: since the construction process for a node is iterated continuously, newly added particles join the node just as though it were being constructed for the first time. Their addition can only cause pre-existing nodes to become closer to the center, so none of the above results are threatened.

7 Experiments

I have implemented the PersistentNode algorithm and demonstrated that it behaves as expected in an amorphous matrix with 2000 particles and communication radius 0.04 (Illustrations for this paper are mainly screen-shots of the running simulation). Traces of experiments demonstrate that when transformed by PSynchDelay, the PersistentNode algorithm is in fact atomic.

Below is a typical trace output from a short experiment. In this experiment, the user creates a node called "bob" with radius 3, then performs a series of six reads and six writes on the node, some simultaneous, some not. In the course of this experiment, which set $T_s = 10$, the node "bob" drifted approximately one diameter (6 hops) from its starting location, towards a denser area of particles, and later operations were performed on particles which were not originally in the node.

Some notes to help interpret the trace: some logging output in the trace has been omitted or reformatted for readability. Each line begins with the particle's time, followed by its ID, a long random number. Events are either invocations (denoted by `recv UI msg`) or responses. The first command, `init bob`, corresponds to invoking $CREATE(bob, 3, \Phi_0)$, the $READ(bob, X)$ invocation is `get bob`, and the $WRITE(bob, X)$ invocation is `set bob X`. Finally, this run was performed with $L = 1.0$, $r = 3$, and two ticks per round, so responses generally follow invocations by $2(3Lr) = 18$, usually plus one from rounding.

```
Creating processors..
6:   0.26506592968033516 recv UI msg: init bob [radius=3, phi=default]
18:  0.26506592968033516 recv UI msg: set bob A [V=(1, 0.24924755367822515)]
21:  0.2040348104571721 recv UI msg: get bob
28:  0.4635829551108004 recv UI msg: set bob B [V=(2, 0.5452059103564912)]
37:  0.26506592968033516 Write completed
39:  0.2040348104571721 Value stored in bob: A
43:  0.26810208788636625 recv UI msg: get bob
47:  0.4635829551108004 Write completed
```

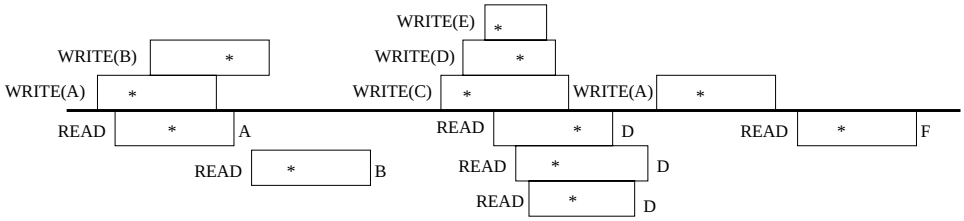


Figure 7: Experiments simulating 2000 particles with communication radius 0.04 produce atomic traces of PersistentNode executing. This diagram shows serialization of the operations in the trace of node "bob", with stars denoting serialization points.

```
61:  0.26810208788636625 Value stored in bob: B
70:  0.4792990984482649 recv UI msg: set bob C [V=(3, 0.7339613509923373)]
70:  0.866065471856458 recv UI msg: set bob D [V=(3, 0.9560773364739286)]
69:  0.17981477442603422 recv UI msg: set bob E [V=(3, 0.9065953621674531)]
75:  0.9542581800945625 recv UI msg: get bob
76:  0.4447567651928277 recv UI msg: get bob
75:  0.26506592968033516 recv UI msg: get bob
88:  0.17981477442603422 Write completed
89:  0.866065471856458 Write completed
89:  0.4792990984482649 Write completed
93:  0.9542581800945625 Value stored in bob: D
93:  0.26506592968033516 Value stored in bob: D
94:  0.4447567651928277 Value stored in bob: D
123: 0.26506592968033516 recv UI msg: set bob F [V=(4, 0.07794895150591896)]
142: 0.26506592968033516 Write completed
149: 0.17405900158905196 recv UI msg: get bob
167: 0.17405900158905196 Value stored in bob: F
```

Analysis of this sequence shows that it is, as expected, atomic. As shown in Figure 7, serialization points can easily be set for the sequence of operations. This holds as well for experiments where large regions of particles are killed off in such a way as to avoid duplicating the node — the typically slower speed of the node means that many failure cases not covered by the above proofs still evince atomicity.

8 Contributions

The PersistentNode algorithm creates distributed READ/WRITE objects that appear locally atomic on an amorphous matrix using only local support within a given radius, and the PSynchDelay transform turns these nodes into atomic objects. When subject to stopping failures, a node reconstructs itself if any member particle survives, and PSynchDelay is atomic or interim atomic if the disruption of the node stays within bounds. The nodes are mobile and move according to a user-specified potential function. This is useful for many problems, including gigascale wireless networking problems such as data storage, congestion control, and partitioning space.

8.1 Motivation: Gigascale Wireless Networking

Wireless networking is expanding rapidly with rising availability and plummeting costs. Consider a scenario a few years hence in which a large city like Boston might have several wireless base stations in every building — a number of nodes on the order of 10^7. If most of the electrical devices in the building are wirelessly networked too, then the total number of nodes could be as high as 10^{10}. If these nodes communicate peer to peer with nearby neighboring buildings, then one could envision the entire city as connected into a packet radio network approximately 10^3 hops in diameter, in which most of the nodes are stationary most of the time. How can one produce useful behavior in this sort of gigascale localized organic network?

8.1.1 Congestion Control

If we set Φ to be $\frac{1}{n_R}$ where n_R is number of reflectors that a particle is participating in, then we have a congestion control function. Nodes operating with this potential function will tend to diffuse towards regions containing fewer nodes, while still avoiding edges of the matrix. Experiments in simulation show that this rapidly converges to an evenly dispersed pattern.

8.1.2 Space Partitioning

Using nodes configured for congestion control, we can fill space evenly. If nodes suicide when a particle is a member of another node simultaneously, and particles which are not in the umbra of any node occasionally create new nodes, then the congestion control will result in a space-filling pattern where every particle is within the umbra of at least one node, and nodes are at least $2r$ distant from each other. If every particle declares membership with the closest node center, then space is partitioned into roughly even units, and if particles use some hysteresis in the decision-making process, the partitioning will change only slowly. If this process is run concurrently with nodes of exponentially increasing radius, then the partitioning can be turned into a balanced hierarchical tree. Experiments in simulation show that the tree thus produced changes extremely slowly at more significant levels of the hierarchy.

8.1.3 Data Storage

With congestion control and space partitioning, data can be stored in the amorphous matrix dispersed along a balanced tree. Control structures in the tree can then allow elimination of duplicate nodes for large disruptions, permitting wider guarantees of atomicity.

8.2 Related Work

Atomic objects are commonly implemented by quorum systems, as in the case of the ABD algorithm[1] and the VitanyiAwerbuch algorithm[18]. Geographically local networks are commonly dealt with in the field of ad hoc networking, and recent research has intersected these two areas. The RAMBO algorithms (Introduced by Lynch and Schvartsman in [14] and extended recently in [6]) manage atomic objects in a (possibly changing) network, but do not directly deal with geographic locality issues. Stojmenovic and Pena have built a geographically aware system[17], but make the strong assumption of supplied coordinates. Dolev, Schiller and Welch[5] have a random-walk based system for establishing groups, although scaling to large networks is impractical given the need for an agent to traverse nodes one by one in a random walk.

9 Acknowledgements

Thank you to Tim Shepard for the idea of applying amorphous computing to gigascale wireless and the Boston scenario. Thanks to Nancy Lynch for advice and support on distributed algorithms theory and proofs, and Seth Gilbert for briefing on quorum systems. Last, but not least, thanks to Hal Abelson and Gerry Sussman for editing and for challenging questions.

References

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

[2] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman and R. Weiss. Amorphous Computing. AI Memo 1665, August 1999.

[3] Daniel Coore. Establishing a Coordinate System on an Amorphous Computer. MIT Student Workshop on High Performance Computing, 1998.

[4] Daniel Coore, Radhika Nagpal and Ron Weiss. Paradigms for structure in an amorphous computer. MIT AI Memo 1614.

[5] Shlomi Dolev, Elad Schiller, Jennifer L. Welch. Random walk for self-stabilitzing group communication in ad hoc networks. PODC 2002: 259.

[6] Seth Gilbert, Nancy Lynch, and Alex Shvartsman. RAMBO II: Rapidly Reconfigurable Atomic Memory for Dynamic Networks. To appear in Proceedings of the International Conference on Dependable Systems and Networks (DSN), San Francisco, CA, June 22nd - 25th, 2003.

[7] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-Serializable Data Services. Theoretical Computer Science, 220(1):113–156, June 1999. Special Issue on Distributed Algorithms.

[8] Roger Khazan. Formal Design and Analysis of a New Virtually Synchronous Group Communication Service for Wide Area Networks. PhD Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 2002.

[9] L. Kleinrock and J. Silvester. Optimum transmission radii for packet radio networks or why six is a magic number. *Proc Natl. Telecomm. Conf.* pp 4.3.1-4.3.5, 1978

[10] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, July, 1978.

[11] Leslie Lamport. On interprocess communication – parts I and II. *Distributed Computing*, 1(2):77-101, April 1986.

[12] Victor Luchangco. Memory Consistency Models for High Performance Distributed Computing. PhD Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 2001.

[13] Nancy Lynch. Distributed Algorithms. Morgan Kaufmann Publishers, Inc, San Francisco, California, 1996. Chapter 8, pages 199-234.

[14] Nancy Lynch and Alex Shvartsman. RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. In D. Malkhi, editor, Distributed Computing (Proceedings of the 16th International Symposium on DIStributed Computing (DISC) October 2002, Toulouse, France), volume 2508 of Lecture Notes in Computer Science, pages 173-190, 2002. Springer-Verlag.

[15] Radhika Nagpal. Organizing a Global Coordinate System from Local Information on an Amorphous Computer. MIT AI Memo 1999

[16] Radhika Nagpal and Daniel Coore. An Algorithm for Group Formation in an Amorphous Computer. Intl Conf on Parallel and Distributed Computing Systems (PDCS'98), 1998

[17] Ivan Stojmenovic and P.E.V. Pena, A scalable quorum based location update scheme for routing in ad hoc wireless networks, SITE, University of Ottawa, TR-99-09, September 1999.

[18] P.M.B. Vitanyi, and B. Awerbuch. Atomic shared register access by asynchronous hardware. Proceedings of the 27th IEEE Symposium on Foundations of Computer Science, 1986.